

### 版权相关注意事项：

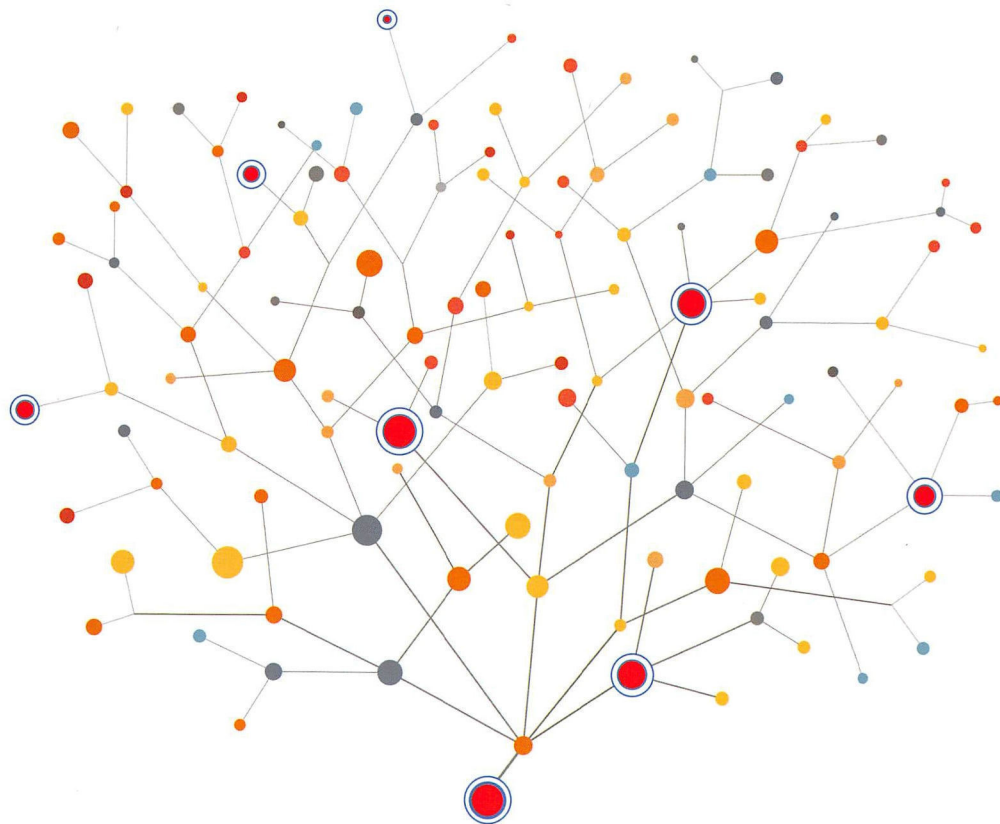
- 1、书籍版权归著者和出版社所有
- 2、本PDF来自于各个广泛的信息平台，经过整理而成
- 3、本PDF仅限用于非商业用途或者个人交流研究学习使用
- 4、本PDF获得者不得在互联网上以任何目的进行传播
- 5、如果觉得书籍内容很赞，请一定购买正版实体书，多多支持编写高质量的图书的作者和相应的出版社！当然，如果图书内容不堪入目，质量低下，你也可以选择狠狠滴撕裂本PDF
- 6、技术类书籍是拿来获取知识的，不是拿来收藏的，你得到了书籍不意味着你得到了知识，所以请不要得到书籍后就觉得沾沾自喜，要经常翻阅！！经常翻阅
- 7、请于下载PDF后24小时内研究使用并删掉本PDF



知名专家联袂推荐，实力专家联合撰写，全面性、透彻性毋庸置疑  
深度讲解区块链核心技术、平台与应用开发，涵盖架构、共识、加密、P2P、比特币、以太坊、Hyperledger、EOS、潜力框架、问题与测评等

# 区块链 核心技术与应用

邹均 于斌 庄鹏 邢春晓 等 著



机械工业出版社  
China Machine Press



## 作者简介

**邹均** 广电运通区块链科技有限公司 CEO、中关村区块链联盟副秘书长。主编技术畅销书《区块链技术指南》，在领先的国际会议和期刊上发表论文 20 余篇，其中区块链论文获 IEEE ICWS 最佳论文奖，共识算法论文由国际期刊《Transaction on Service Computing》收录并刊登。曾荣获澳中校友会“杰出校友奖”、麦考瑞大学“校长奖”。

**于斌** 北邮在线教育投资集团总裁、中国电子学会区块链专委会委员、中关村区块链产业联盟专家，是上海财经大学，亚洲财经商学院特聘教授。北京邮电大学通信与信息系统专业博士，主编《金融科技概论》等专著 4 本，曾获得国家科技进步二等奖，教育部一等奖。网络教育、金融科技、区块链等领域专家。

**庄鹏** IBM 全球服务金融服务部高级顾问经理、资深架构师。14 年金融行业架构设计与战略咨询规划经验。拥有服务转型、大型企业级分布式系统架构设计、大数据分析、金融支付方面的丰富实施经验。最近三年专注于区块链和分布式账本架构研究，区块链相关应用和数字货币咨询研究，多次作为区块链峰会的讲师、培训专家。

**邢春晓** 清华大学信息技术研究院和互联网产业研究院副院长，主要研究领域：计算机软件与理论、数据库和数据仓库、大数据管理和分析、知识工程和软件工程、区块链与数字经济、智慧城市（政务、商务、文化和医疗健康）等领域。发表学术论文 350 余篇，其中 SCI 40 余篇、EI 150 余篇，发明专利 40 项。







华章IT  
HZBOOKS | Information Technology



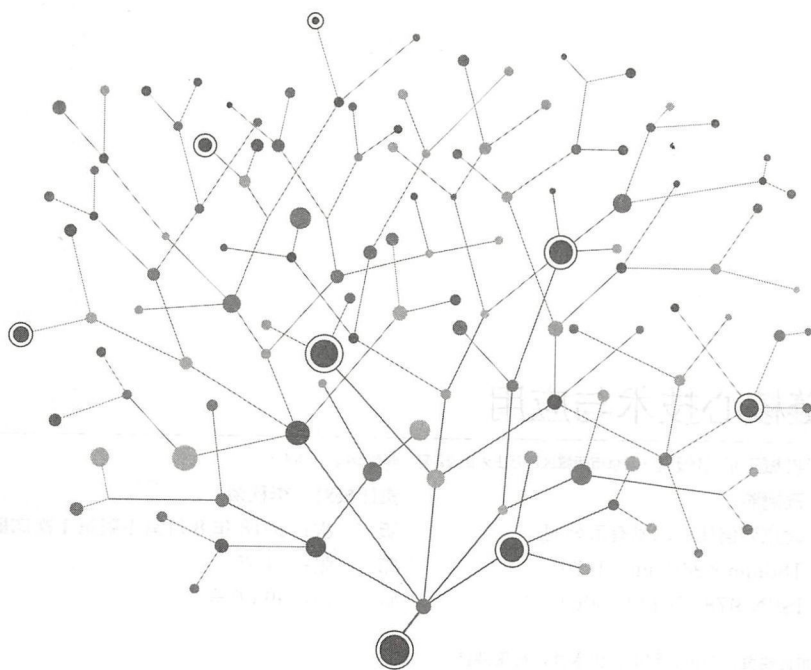




区块链  
技术丛书

# 区块链 核心技术与应用

邹均 于斌 庄鹏 邢春晓 等 著



机械工业出版社  
China Machine Press



## 图书在版编目 (CIP) 数据

区块链核心技术与应用 / 邹均等著. —北京: 机械工业出版社, 2018.8  
(区块链技术丛书)

ISBN 978-7-111-60614-7

I. 区… II. 邹… III. 电子商务 - 支付方式 - 研究 IV. F713.361.3

中国版本图书馆 CIP 数据核字 (2018) 第 172264 号

## 区块链核心技术与应用

出版发行: 机械工业出版社 (北京市西城区百万庄大街 22 号 邮政编码: 100037)

责任编辑: 高婧雅

责任校对: 李秋荣

印 刷: 北京诚信伟业印刷有限公司

版 次: 2018 年 8 月第 1 版第 1 次印刷

开 本: 186mm×240mm 1/16

印 张: 24.25

书 号: ISBN 978-7-111-60614-7

定 价: 99.00 元

凡购本书, 如有缺页、倒页、脱页, 由本社发行部调换

客服热线: (010) 88379426 88361066

投稿热线: (010) 88379604

购书热线: (010) 68326294 88379649 68995259

读者信箱: hzit@hzbook.com

版权所有·侵权必究

封底无防伪标均为盗版

本书法律顾问: 北京大成律师事务所 韩光 / 邹晓东





## About the Authors 作者简介

邹均，广电运通区块链科技有限公司 CEO、中关村区块链联盟副秘书长。主编技术畅销书《区块链技术指南》，在领先的国际会议和期刊上发表论文 20 余篇，其中区块链论文获 IEEE ICWS 最佳论文奖，共识算法论文由国际顶级期刊《Transaction on Service Computing》收录并刊登。澳大利亚麦考瑞大学计算机博士、商学院 MBA，曾荣获澳中校友会“杰出校友奖”、麦考瑞大学“校长奖”。

于斌，北京邮电大学通信与信息系统专业博士。主编《金融科技概论》等专著四本，曾获得国家科技进步二等奖，教育部一等奖。是网络教育、金融科技、区块链等领域专家，现任北邮在线教育投资集团总裁、中国电子学会区块链专委会委员、中关村区块链产业联盟专家，上海财经大学、亚洲财经商学院特聘教授。

庄鹏，现任 IBM 全球服务金融服务部高级顾问经理，资深架构师。拥有 14 年金融行业业务和应用架构、IT 系统集成、应用系统开发和管理、战略咨询规划经验。特别是在金融行业面向服务架构转型，企业服务总线规划和实施，分布式系统架构，大数据分析，金融支付应用方面具有深厚的技术功底及实施经验。最近三年专注于区块链和分布式账本架构研究，区块链相关应用和数字货币咨询研究，多次担任区块链峰会讲师、培训专家。

邢春晓，清华大学信息技术研究院副院长，清华大学信息技术研究院 Web 与软件技术研究中心主任，清华大学智慧城市大数据研究中心主任，中国计算机学会副主任与专委委员，IEEE 和 ACM 会员。主要研究领域为数据库、区块链、数据和知识工程、海量数字媒体管理等。发表学术论文 300 余篇，其中 SCI 40 余篇、EI 150 余篇，软件著作权 23 项，发明专利 40 项，教育部科技成果 1 项。多次作为主要负责人和技术骨干参与国家重点科技项目。

张海宁 (Henry Zhang)，VMware 中国研发中心技术总监，加拿大西蒙弗雷泽大学计算机科学硕士，Harbor 开源企业级容器镜像仓库创始人，超级账本 Cello 项目贡献者，Cloud



Foundry 中国社区早期技术布道师之一，“亨利笔记”公众号作者。目前着重关注企业区块链应用、容器和云计算等领域的研究和开发工作。

**蒋勇**，技术畅销书《白话区块链》作者。专注于分布式系统设计，10 年企业信息化经历。2012 年开始接触比特币及其相关技术，熟悉区块链 1 代技术（比特币）、2 代技术（以太坊、超级账本），并进行过源码级原理研究，目前在进行智能合约安全编译以及多链架构的研发设计。

**唐屹**，教授，中山大学博士，广州大学数学与信息科学学院信息科学系主任。曾访问美国北卡罗来纳州立大学、香港浸会大学等高校。专注于区块链安全与应用、网络信息安全、分布式计算等领域的研究，为国外知名安全公司开发过椭圆曲线密码软件，获密码科技进步二等奖（省部级）。主持或参与完成多项国家级及省部级项目，在国内外学术期刊和会议上发表学术论文多篇。

**邵周**，中国计算机学会区块链专委会委员、中关村区块链联盟金融专委会专家、TOGAF 认证企业架构师、信息安全与风险管理专家，是以结果为导向的技术领导者，也是较早一批关注和实践物联网、区块链等技术的布道者和践行者。研究方向有高性能区块链、分布式存储、分布式算力、可衡量注意力、跨链协同、加密资产锚定等，著有数本科技书籍。目前就职于亚洲基础设施投资银行。

**郭莹城**，IBM 高级软件架构师、咨询师、敏捷开发技术教练、极客、登山爱好者。11 年电信、金融、电子政务软件研发经验，参与了新一代深圳证券交易所交易系统，以及多个外资银行的核心系统研发，对 Lisp 编译器有研究与心得，精通 Java、Scala、Go、Python、Ruby、Lisp 多种编程语言，Hyperledger 与以太坊智能合约研究者，区块链 P2P 算法专家，IBM 区块链研究小组成员。

**刘胜**，联动优势科技有限公司首席架构师、中国电子学会区块链专委会委员、可信区块链联盟副理事长。承担国家级区块链和数字货币等课题的研究，参与《可信区块链》《支付清算行业可信区块链》等标准编写。20 余年移动支付、数字证书认证、安全支付、区块链等领域一线研发和底层架构经验，带领团队自主研发针对行业联盟链场景的区块链底层框架 UChains（优链）。提交并公开发明专利 50 多项，其中区块链专利 8 项，已授权发明专利 5 项，曾获 2015 年北京市科学技术三等奖。

**范金刚**，食品区块链、金融区块链和能源区块链行业专家，太一云技术股份有限公司常务副总裁、中国区块链生态联盟副理事长、中国电子学会区块链专委会执行秘书长。曾任中关村区块链产业联盟副秘书长。主持开展过区块链基础平台测评工作，组织并策划了第一届中国区块链技术创新应用大赛等活动。2016 年在《电力信息与通信技术》杂志上发表学术论





文《区块链在能源互联网中的应用》。

**张桂刚**，清华大学博士后、中国科学院自动化研究所副研究员、研究生导师。主要从事人工智能、大数据及区块链研究。出版专著 1 部，发表 SCI/EI 论文 60 余篇。

**陈家豪**，广州大学硕士。从 2016 年开始接触区块链，在读期间主导通用加密货币钱包的开发、区块链网络安全分析等。参与 VMware 公司区块链即服务项目 BOV（Blockchain On Vsphere）开发，是 Hyperledger 社区 Cello 项目的代码贡献者之一。擅长虚拟化、区块链安全、密码学应用等技术，熟悉区块链平台比特币、以太坊、超级账本并有相关开发经验。

**张权**，本书特约策划人，北邮在线区块链教育与研究中心主任、信息化与数字经济研究中心（IDER）联合创始人兼 IDER 学院院长、英国 CCEG 区块链运营主管，中国移动通信联合会国际区块链创新应用联盟秘书长助理。关注区块链相关的教育与研究方向及搭建专业实践平台构建人才培养体系；擅长区块链、人工智能、物联网、制造业等项目的产品营销、市场业务拓展及平台建设及运营工作。



## 序一 处于“十字路口”的区块链技术及其应用<sup>①</sup> *Foreword 1*

区块链和与之相联系的加密数字货币，无疑是 21 世纪“横空出世”的新事物。过去数年，特别是 2017 年以来，区块链技术受到全球范围内的科学家、思想家、经济学家、企业家，以及政府、社会和经济组织的关注、参与和应用，全方位地影响了现存的经济结构和经济行为，一方面，迅速形成了全球性的“区块链”运动；另一方面，因为区块链技术的先天缺陷和现实困境而陷入“十字路口”境地。现在，到了以理性态度理解和认识区块链与加密数字货币真实状况，突破区块链创新瓶颈，实现区块链技术体系进化的时候。正是在这样的历史性期待下，邹均及其他几位作者撰写的《区块链核心技术与应用》，从剖析区块链的核心技术入手，对区块链的技术体系进行了理性的梳理，触及区块链的深层结构和科学基础，所以我给予相当的肯定。本书共 12 章，26 万字。作为篇幅有限的序言，主要集中讨论处于“十字路口”的区块链及其技术体系所面临的 4 个基本问题。

1. 如何理解和认识区块链所存在的“先天缺陷”，以及区块链技术背后的科学层面。一般来说，任何一项新技术都可能存在“基因”和“染色体”问题，区块链也不例外，很可能反映在以下的 4 个“局限性”。

1.1 囿于数学工具的局限性。通常认为，区块链的核心技术是“密码学”，而区块链的密码学的重点之一则是哈希函数。也就是说，哈希函数是区块链的基石之一。哈希函数的种类很多，大多数哈希函数都是迭代性的，即使用一个哈希函数，用不同的参数进行多次迭代运算。问题是，哈希函数及其紧密联系的“素数”，甚至现代密码学，最终根源于纯粹数学分支之一的“数论”。高斯说过：“数学是科学的皇后，数论是数学的皇后”。因为数论还在发展，哈希函数尚属年轻，区块链还存在诸如算法选择不当，造成较多碰撞，导致性能下降等问题，

<sup>①</sup> 在本文的撰写过程中，数学专家崔巍、齐洪胜和计算机技术专家邹均、程文彬、张洪为，特别是郑延伟，深入参与了本文的技术讨论，提供了专业意见和建议，在此表示感谢。



加之对哈希函数的有限移植,意味着区块链的底层技术建立在还处于新生阶段的“地质板块”上。至于, Merkle 树因为与哈希函数的内在联系也归结于数学问题,所谓的拜占庭将军问题的本质更是一个数学问题。在这样的意义上,可以试图用数学语言将区块链描述为:以数论为基础,通过哈希函数实现的一种“复合函数”构造。

**1.2 囿于“博弈论”的局限性。**区块链的本质是一种多方参与,且形成平衡关系的共识系统。这也是区块链,特别是公有链存在价值的关键所在。可以将共识系统理解为一种节点之间的“均衡”,建立在“博弈论”基础上的“纳什均衡”最接近反映区块链共识系统的状态。具体来说,纳什均衡是指这样的一种策略组合:在一个非合作博弈过程中,任一博弈方只有选择某个确定性策略,才能获得最佳收益。如果任一博弈方单独选择变换策略,悖离纳什均衡,都会损害自己的收益。问题是,当年诺伊曼和纳什研究的是有限“节点”下的小规模博弈,早已经不足以面对“由几十亿节点的庞大对象构成的社会、经济等复杂行为”<sup>①</sup>。MIT 的一个近期成果是,一位计算机科学博士在其论文中指出:“找到纳什均衡点是几乎不可能的事”<sup>②</sup>。区块链和博弈论,包括纳什均衡的现实关系是:一方面,区块链需要博弈论,包括纳什均衡工具的支持;另一方面,区块链节点的算术级数,甚至几何级数发展模式,已经突破了博弈论的框架和体系。总之,因为区块链的节点无限扩大,所以支撑区块链的博弈论和纳什均衡必然捉襟见肘,出路何在,至今并没有找到最终的科学路径。

**1.3 囿于计算机语言和代码的局限性。**区块链通过计算机语言和代码进行技术实现,没有软件的注入,就没有生命力和运行可行性。但是恰恰不存在完美的软件。

其一,区块链编程语言多元化,难以找到占有绝对优势的区块链编程语言,只能通过不同编程语言的互补性加以改善。在现实中,很可能发生因为任何一种编程语言自身不足,以及不同的编程语言不足的叠加,对现有区块链造成本源性的伤害。

其二,在现阶段,区块链编程语言主要依赖 C++、Java、Go 等几种“高阶语言”,但是,这些语言都需要演进,以求满足区块链技术实现的需求。可以确定的是,现有的“高阶语言”仍有很大的改进空间。逻辑上说,整个计算机语言体系仍会继续发展,新一代计算机语言势必对区块链产生冲击和影响,推动区块链的演进。

其三,现有的计算机语言正在面临与其他新技术的融合,进而影响区块链的技术体系。例如,人工智能技术和计算机语言的融合,很可能引发计算机语言系统的变革。

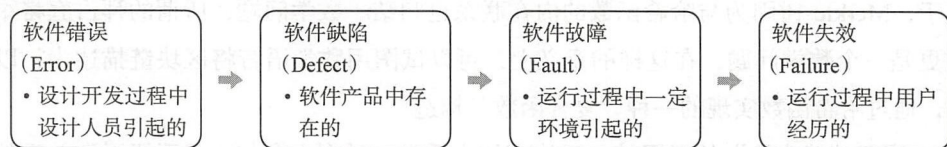
其四,编程人员的自觉和非自觉的错误。本书提供了区块链与软件相关的错误、缺陷、

① 见本书 12.5 节。

② 见本书 12.5 节。



故障和失效的关系框架图<sup>⊖</sup>。



1.4 囿于预期时间实验长度的局限性。至今，区块链的实验历史相当短暂，支撑比特币的区块链历史最长，应用也不足十年。但是，依据比特币系统的设计和比特币的算法，推测直到 2140 年最终产出 2100 万单位，即使从 2018 年起计算，仅仅挖全部的比特币，还需要 122 年的时间。从理论上说，比特币技术系统的使用寿命没有上限，这个过程中，比特币系统的运行不可中断，甚至不允许进行实质性的修改。所以，本书提出了如何保证比特币系统在未来 122 年安然无恙地运行问题，是有相当意义的。其实，岂止比特币的区块链技术系统，绝大多数区块链的设计和應用并非是短期内可验证的，现在还没有足够的案例证明，现行支持区块链的软件和硬件系统能够支持长周期的时间目标。

进一步说，到目前为止，支撑区块链的主要底层技术的产生，早于区块链。之后，因为有了区块链理念，这些技术得以重新组合。也就是说，现阶段的区块链技术，及其数学和科学的基础，还是相当脆弱的，难以支持在商业、经济和社会的长期和大规模的应用。在人类现代科学技术发展的过程中，从基于有限科学原理的技术尝试起步，最终形成完整的学科和科学体系，不乏历史案例。例如，莱特兄弟制造飞机之时，所依据的不过是一些初步的科学原理和技术，但是伴随飞机的进化和工业化，最终形成了以飞机整体设计为目标，包括航空学、材料学、电子科学、工程制造学等众多学科的综合科学技术体系。

所以，现阶段区块链的技术很像莱特兄弟飞机的试验阶段。要想全面实现区块链的理念，就像飞行器的历史，最终不仅要实现在全天候和全方位的天空飞行，而且最终要进入宇宙空间，需要的是一个完整的科学和技术体系的支持。可以这样认为，因区块链所组合的技术，还需要一个适应区块链广泛应用的调整时期，或者进一步开发的时期，最终区块链可能演变出一个科学含量极高的综合学科体系。

2. 区块链是否存在现实的和潜在的威胁？区块链面临着为数众多的技术性困境，诸如“可扩展性”技术，“隐私保护”技术，“存储”技术，等等。但是，真正构成对区块链体系现实和潜在威胁的很可能集中在以下几个方面。

2.1 共识层结构性失衡的可能性。在区块链的架构中，或者在区块链的分布式系统中，共识层至关重要。共识的本质是算法，一个严格共识算法需要满足 4 个条件：终止性、一致

⊖ 见本书图 12-3。

性、合法性、诚实性。<sup>①</sup>从技术层面来讲，区块链的共识算法是基于异步通信场景。这样就要涉及 FLP 定理。该定理的研究对象覆盖异步通讯的没有时钟、不能时间同步、不能使用超时、不能探测失败、消息可任意延迟、消息可乱序等一系列特征。在异步通讯场景下，即使只有一个进程失败，也没有任何算法保证非失败进程达到一致性。<sup>②</sup>在这样的情况下，通常的共识算法会倾向选择安全性并牺牲活性，难以保证在有限时间内达成共识<sup>③</sup>。也就是说，现在的各类区块链的共识算法只有相对意义，并不存在统一的共识算法，唯有根据不同的需求，不同类型的区块链采用相对适宜的共识算法。<sup>④</sup>

| 诚实性 | 拜占庭容错 | 终止性 / 确定性 | 常见共识算法 | 典型场景                  |       |
|-----|-------|-----------|--------|-----------------------|-------|
| 1   | 否     | 否         | 确定性一致  | Paxos、RAFT 等          | 企业私有链 |
| 2   | 是     | 是         | 确定性一致  | PBFT、Q/U、Zyzyva、RBF 等 | 行业联盟链 |
| 3   | 一定程度  | 是         | 大概率一致  | PoW、PoS、DPoS 等        | 公有区块链 |

这样，就引申出两个问题：其一，已有的共识算法如何应对当下区块链数量爆炸，类型组合多样化，以及区块链类型之间差异增加的情形；其二，区块链是特定的多维动态系统，即多中心或无中心、更加分布式的网络。这个网络跨越多个子网、多个数据中心、多个机构、多个运营商，甚至多个国家，如果区块链联盟进入实质化，怎样协调不同共识算法，实现共识的共识？

不久前，一位以 maxdeath 作为笔名的作者提出：区块链的第三代技术突破，一个是零知识验证技术，另一个可能的突破点是真正无限扩展的共识算法。<sup>⑤</sup>这个看法，颇值得注意。

还有，区块链的功能依赖于去中心化账本，而去中心化账本取决于不同节点上的账本数据的一致性和正确性，最终取决于分布式系统中实现状态共识的算法。因为区块链的算法对一致性的制约，不可避免地影响了去中心化账本的深层基础及区块链的功能。

**2.2 算力的垄断倾向和趋势。**区块链的运行机制可以理解成不断将“单一或多个输入值”转换为“单一或多个结果”的过程，因此区块链是一种“计算”<sup>⑥</sup>，一般来说，区块链天然需要算力的支持。特别是，以比特币为代表的区块链技术，使用 PoW<sup>⑦</sup>来确定记账权。这一机制要求节点进行大量的复杂函数计算，使得记账节点完成共识的同时，增加对区块链的攻击成本，提高了安全性。引入这一机制对区块链技术的发展产生了深远的影响：唯有通过

① 见书内文 12.6 节。

② 见参考资料《FLP Impossibility》，<https://blog.csdn.net/ch77716/article/details/27963079>。

③ 见书内文 5.1.2 节。

④ 见书内文表 12-4。

⑤ 见参考资料 <https://www.zhihu.com/question/265273597/answer/294404050>。

⑥ 见参考资料 <https://zh.wikipedia.org/wiki/%E8%AE%A1%E7%AE%97>。

⑦ 见书内文“5. 共识算法”。



算力不断挖出区块来实现记账。如果区块链没有算力，就无法产生新的区块，就会出现交易没有人去记账的情况。这就意味着区块链的“死亡”。算力的作用也开始被狭义化为“挖矿”，其衡量标准等同于“哈希率”，即计算哈希函数输出的速度。“矿工”开始成为比特币等区块链中的重要群体，在比特币的管理范畴中不断要求更高的权利。问题在于，有可能发生区块链天然算力遭到垄断的情况。以比特币为例，过去的十年间，其算力经历了从 CPU 到 GPU 的个体时代，短暂的 FPGA 阶段，最终进入了通过 ASIC 矿机支持的矿池垄断时代，其标志是比特币全网算力以 PBhash/s 为单位（1PB=1024T，1TB=1024GB，1GB=1024MB，1MB=1024KB）。截至本序写作时，比特币预计全网算力已经将要达到 37 000P<sup>①</sup>。

| 比特币全网基本信息 (自动刷新:30秒) |                   |         |                               |        |                    |        |                 |
|----------------------|-------------------|---------|-------------------------------|--------|--------------------|--------|-----------------|
| 货币名称                 | 比特币/Bitcoin/BTC   | 预计全网算力  | 36,397 PH/s                   | 已开采BTC | 17,070,013         | 未开采BTC | 3,929,988       |
| 总市值                  | \$133,789,636,971 | Block总数 | 525,601                       | 24h开采  | 171 块<br>2,100 BTC | 平均1h开采 | 7.1 块<br>88 BTC |
| 当前难度                 | 4,306,949,573,981 | 预计下次难度  | 4,843,776,149,929<br>(+12.5%) | 难度调整   | 574 块以后            | 预计需时   | 3天8小时           |

期间，基于去中心化的分布式记账的比特币，原本与之配合的分散和自由竞争的算力，迅速遭到包括芯片技术升级、能源、人力、资本大量投入的组合型冲击，造成以算力日益垄断为特征的异化。从技术角度说，在区块的挖出速度不能改变前提下，算法就会根据算力增大而提高和控制区块挖出的难度，当全网算力足够大的时候，挖矿的难度也就足够大，任何个人不太可能独自挖出完整的区块，只有加入矿池，才可能以矿池的算力挖出完整区块，个人按照其贡献算力的比例去分成。不仅如此，在矿池环境下，因为全网挖矿难度的调整不是实时的，如果发生矿池突然中止挖矿，会导致剩余矿工无法在当下的挖矿难度下准时产生区块，最终造成区块链“死亡”式的“同归于尽”。所以，建立在强大“算力”基础上的矿池实质上绑架了区块链。先是比特币，进而以太坊也面临相同的境地。

如果说中本聪是智者，却没有想到基于资本和工业能力所支持的算力垄断，早已经不仅仅是一个简单的能源、人力、资本的竞争和消耗问题，也不仅仅是个人通过挖矿获得比特币财富的权利遭到剥夺的问题，而是一旦处于垄断状态的算力失去控制的机制，很可能构成对区块链和加密数字货币生态的毁灭性破坏的问题。现在很可能只有两种可能性：①为了维持算力垄断，成本不断增大，当成本高于收益的时候，就会停止；②为了维持算力垄断“竭泽而渔”，最终同归于尽。前者可以理解为“软着陆”，后者可以理解为“硬着陆”，现在显现的主要可能性是后者。不论是“软着陆”还是“硬着陆”，都需要警惕，更需要通过技术底层创新加以防范。

需要说明的是，目前基于 PoW 所衍生出来的种种弊端，人们已经尝试了若干不同的替代

① 参考图片资料来自 <http://mining.btcfans.com/>。

方案,例如 PoS、DPoS<sup>①</sup>等。特别是, Vitalik Buterin 针对“联合化的矿工团体出于自利目的将算力集体转移以攻击旧区块链 (spawn-camp)”的可能性,提出可以加强从客户端出发的验证机制,利用均匀分布的个体用户的“协调性问题”,增加上述攻击成功的难度<sup>②</sup>。但是,以上种种方案并不意味着 PoW 的思想已经过时,也不意味着“算力”应当被摒弃。其如何演变,以及 PoW 究竟会如何发展,“算力”是否能够在区块链中发挥更重要的作用,还需要时间的验证。

**2.3 传统经济模式和工具的侵蚀。**面对区块链及其所支持的加密数字货币的横空出世和蓬勃发展,有一种广泛传播的乐观看法:区块链和加密数字货币诞生、传播与应用,具有强大的生命力,代表了打破传统经济生态平衡,改变传统金融和产业秩序的“新物种”。但是,现实更多展现的是相反的一面,即传统经济模式与工具对区块链和加密数字货币的生态体系影响。

其一,包括比特币在内的主要加密数字货币的价值,需要通过以美元为代表的法币加以体现。之所以发生这样的情况,主要因为在现实世界还没有形成比特币或者其他加密数字货币的自我循环和交易的市场。

其二,传统以法币作为投资手段的各种资本,全面进入区块链和加密数字货币领域,形成“资本主权”。

其三,诸如商业银行、投资银行、交易所等机构模式,正在与区块链、加密数字货币和 Token“嫁接”,甚至“融合”。

其四,部分“移植”区块链和加密数字货币原理和技术,以求强化传统经济和货币体系。一些国家开发的法定加密数字货币,就是典型案例。

要特别注意的是区块链“财富效应”幻觉,夸大以区块链为技术基础的数字资产的价值转移、定价与交易,误导人们以传统商业化的原则理解和判断区块链的价值。简言之,当下回答如何看待传统经济模式和区块链系统的相互作用,或者彼此混合的后果,是否能够重构人类信任体系,最终谁改造谁,还为时过早。但是,有一点是值得考虑的:区块链及其理念与技术原本是为了解决传统经济制度的缺陷,然而面对传统经济工具、制度和机构的全方位接触,尚处于早期的和脆弱状态的区块链生态,并不具备强大的自我保护机制,面临被传统经济模式和工具再改造的可能性。

**3. 现阶段区块链存在怎样的创新路径?**区块链过去、现在和未来的生命力,都取决于其创新能力。区块链的创新主要是两类:在现阶段区块链主流架构下的创新,以及从本源上突

① 见书内文“5. 共识算法”。

② 见参考资料《Engineering Security Through Coordination Problems》, Vitalik Buterin。



破了主流框架的创新。

3.1 基于区块主流架构和具有“路径依赖”特征的创新。现在普遍看法是，比特币是区块链的 1.0，以太坊是区块链的 2.0。在主流框架下的创新主要集中在以下几个方面。

其一，突破“可扩展性”限制。主要包括比特币通过“分叉”的扩容方案，以太坊自身的 Plasma、State Channel、Raiden、Truebit、Sharding 和 Casper 等扩容方案<sup>①</sup>，以及诸如侧链技术（RootStock、Polkadot、Cosmos）、区块扩容、链下计算、分区共识、分片的“内部分割”等处于试验阶段的选择方案<sup>②</sup>。

其二，改善存储。工程实例至少有：Swarm（以太坊的 P2P 文件共享协议）、Storj（一种“分布式存储”技术），以及 IPFS（一种 P2P 超媒体协议）<sup>③</sup>。

其三，EOS 代表的区块链操作系统。根据 EOS 白皮书显示，预计其可扩展至单链几千 TPS、全网并行百万 TPS 的交易吞吐量，并且主网已于 2018 年 6 月上线，是未来区块链平台强有力的竞争者。<sup>④</sup>

其四，开发具有隐私和法规的区块链，例如 Cardano。Cardano 是世界上第一个由研究为主导，基于科学哲学开发出来的区块链项目，也是第一个采用同行评审技术的区块链项目，可用于发送和接收数字资金，支持各种去中心化应用和智能合约。<sup>⑤</sup>

其五，提高智能合约的安全性。形式化验证技术正在逐步应用到区块链智能合约代码检查。其原理是，根据某个或某些形式规范或属性，使用数学的方法证明其正确性或非正确性。<sup>⑥</sup>

其六，满足区块链测试需求，参考“ISO/IEC 25010 标准”来完善测试模型。在这本书提供的参考测试模型中，区块链评测涉及 8 个维度，31 个分维度。<sup>⑦</sup>

3.2 突破现阶段区块链主流架构的创新。必须注意到，在区块链主流架构下的创新，只具有相对意义。例如，比特币在顶层设计阶段就没有彻底解决共识问题，只是将问题加以转换，“一方面通过区块链的序号作为虚拟时间基准，另一方面通过‘挖矿’的经济动力来促使比特币链的不断延伸”<sup>⑧</sup>。在这个意义上，中本聪用此类经济方法解决分布式系统共识问题确实相当智慧。进一步说，现阶段的区块链，难以升级。一旦区块链被部署和进入生产模式，

① 见书内文 10.3.1 节。

② 见书内文 10.2 节。

③ 见书内文 10.3.10 节。

④ 见书内文“10.3.2 节”。

⑤ 见书内文“10.3.3 节”。

⑥ 见参考资料 <https://zh.wikipedia.org/wiki/%E5%BD%A2%E5%BC%8F%E9%AA%8C%E8%AF%81>。

⑦ 见书内文 12.2 节。

⑧ 见参考资料《区块链正本清源：从计算机科学评看区块链的起源和发展》。<https://blog.csdn.net/omnispace/article/details/80467188>。

在功能上进行添加、修改和删除，难度甚大，成本甚高。<sup>①</sup>现在，通常的区块链修改，都会造成区块链系统的软分叉或者硬分叉，造成时间、精力和经济上的浪费。

所以，区块链的真正创新必须突破主流架构，实现顶层理论设计和数学方法上的创新。这种突破主流框架的创新，已经悄然开始。到目前为止，DAG（Directed Acyclic Graph，有向无环图）为代表的技术属于突破现阶段区块链主流架构的创新。“比特币的效率一直比较低，基于 PoW 共识下的出块机制是原因之一，由于链式的存储结构，整个网络中同时只能有一条链，导致出块无法并发执行。DAG 从根本上摒弃了区块概念，交易直接进入全网中，达到所谓的无区块（Blockless）效果，网络中的交易可以容纳 N 倍。”<sup>②</sup>

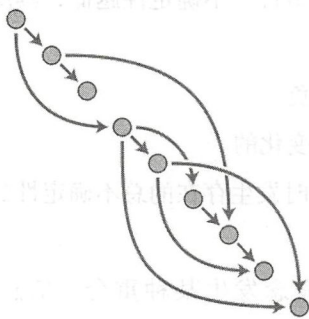
具体说，DAG 有 4 个特点。

其一，交易速度快。交易速度远远高于基于 PoW 和 PoS 的区块链交易速度。

其二，无须挖矿。DAG 把交易确认的环境直接下放给交易本身，无须由矿工打包成区块后同意交易顺序。所以 DAG 网络中没有矿工的角色。

其三，无手续费。交易发起只需要做简单的 PoW，整个网络中的 PoW 都是发起交易者自己做的，而不是交给矿工，所以发起交易无须手续费。

其四，需要见证节点。DAG 需要见证人机制的存在。超越 DPoS、PoS、PBFT，最终实现在效率与安全性上的一种平衡。<sup>③</sup>DAG 的数学基础不再是数论，而是图论，即拓扑学，即以空间、维度与变换作为研究对象的学科。图论原理显然更接近对区块链基本模式的描述。DAG 模式如下<sup>④</sup>。



第一次提出 DAG 跟区块链结合是在 NXT 社区，该社区的成员大多集中在东欧和俄罗斯，去中心化对他们更具有现实意义。现在，基于 DAG 原理和技术的 IOTA，由 4 名平均年龄不

① 见参考资料《MOAC 号称“众链之母”：他们在区块链技术上有何创新？》。

② 见参考资料 <https://zhuanlan.zhihu.com/p/31764777>。

③ 同上。

④ 见参考资料 <https://ethereum.stackexchange.com/questions/1993/what-actually-is-a-dag>。

到 30 岁、最年轻者仅 21 岁的团队所创立，推出不到两年，其价格在虚拟货币中已经攀升至第 8 名，他们更高的目标是成为“物联网”的骨干。此外，Swirls 公司开发的 Hashgraph 技术也是 DAG 结构的一种。Hashgraph 通过 Gossip about Gossip 协议，“让每个节点都维护着所有节点跟其他节点的通信历史，每个节点在完成拜占庭协议时，不需要经过网络多轮通信，节点本地环境就可以直接模拟拜占庭协议。Hashgraph 在数学上可以证明满足异步拜占庭容错，至少跟比特币一样安全。”<sup>①</sup>不过，对于 Hashgraph 技术，尚有很大争议。

4. 怎样实现区块链系统维系“熵值减少”状态？区块链是典型的信息系统，其生命力取决于其内在的信息熵的大与小和高与低。

邹均在本书前言中提出：“在香农创立的信息论中，信息是确定性的一个度量，而熵也是信息量的一个度量。熵越大，越无序，信息量越少。而从这个意义上来说，区块链系统是一个熵值减少的系统，因为共识所确定的状态就是信息，而信息也就是有序和确定性”。<sup>②</sup>邹均的上述说法是否精确，可能还需要讨论。但是，邹均提出了两个有意义的问题：其一，区块链是一个熵值减少的系统；其二，共识导致熵值减少。

4.1 何谓“信息熵”？为了理解“信息熵”，首先需要理解什么是“信息”？信息没有标准定义，按照香农（C. E. Shannon, 1916-2001）的“信息论”，可以理解为一串以逻辑论的 01（即计算机的“二进制”）编码的数列。至于信息熵，则是基于数学，或者统计学上的抽象概念，所描述的是信息本身的一种性质，这种性质独立于信息的形式内容。具体说，信息熵的几个基本属性如下。

1) 单调性。发生概率越高的事件，不确定性越低，其所携带的信息熵越低。相反，事件的可能选择越多，不确定性越大。

2) 非负性，即信息熵不能为负。

3) 信息熵应该是随概率连续变化的。

4) 累加性，即多随机事件同时发生存在的总不确定性的量度可以表示为各事件不确定性的量度的和。<sup>③</sup>

在这里，信息熵与信息量概念发生某种重合。信息量是衡量信息消除的不确定性（Uncertainty）的度量，更准确地说，就是“变量不确定度的平均度量”，信息论中将它称之为

① 见参考资料 <https://www.jianshu.com/p/9f181cefba8d?from=timeline>。

② 见书前言。

③ 参考资料：<https://www.zhihu.com/question/22178202>。



“信息熵”。<sup>①</sup>或者说,信息熵是信息的不确定性的度量,不确定性越大,信息熵越大。信息熵也可以理解为将一个事件用 01 编码的一串数列的平均码长,码长的单位为 bit。所以,信息熵也是一种数学预期。

**4.2 区块链何以形成有序?**从根本上说,区块链的信息系统天然具有克服熵增,实现熵减的结构。因为,一般的信息系统不同于物质世界,具有以下的基本特性:①可复制性,难以唯一;②无序扩散性;③易失真、演变、丢失;④不考虑信源在信息流动过程中扮演的“麦克斯韦妖”的角色,信息熵因为信息的基本特性更容易出现无序态;⑤加之中心化信息系统,本质上只是一个没有任何坚实基础和被放大和降噪的信息管道,难以实现系统的真正有序和确定性。所以,一般的信息系统很难克服熵增属性,难以实现系统的真正有序和确定性。而区块链信息系统,因为包含共识机制,包含时间、主体、内容在内的全息记录,例如时间戳技术,以及包含非中心化系统和节点分布,导致熵减,有序和增加稳定性。

但是在现实的区块链系统中,区块链的熵减情况不是自然发生的。也就是说,实现区块链架构中的任何一个层次的数据和信息的真实性和完整性,绝非易事,特别是存在共识层结构性失衡的可能性。如果考虑到区块链天然需要算力,需要通过不断挖出区块来实现记账,而挖出区块需要耗费日益增加的能源和人力资源的情况,区块链及与其关联的加密数字货币甚至无法摆脱热力学的熵定理。所以,就区块链技术和日益广泛的应用而言,怎样实现持续性的熵值减少状态,具有相当的挑战性。

**4.3 耗散结构理论和区块链。**<sup>②</sup>在物理学上,最能够解释区块链的理论很可能是“结构理论”:一个远离平衡状态下的开放系统,在与外界环境交换物质和能量的过程中,通过能量的耗散过程和系统内部的非线性动力学机制,能量达到一定的程度,熵流可能为负,系统总熵变可以小于零,宏观上可能从原来的无序状态转变为时间、空间或功能的动态有序状态。<sup>③</sup>现在看,区块链很可能属于耗散结构,是一种自组织系统,具有内部各因素之间产生协同动作和相干效应,解决了信息系统从无序走向有序的机制。区块链的自组织系统能够适应普遍的环境,更加柔韧和健壮。特别是,自组织的区块链系统具有自我完善的可能性,消除了宏观上必然熵增引发的衰败趋势。进一步说,区块链作为一种自组织的耗散结构,具有自我意

① 1948年,香农在长达数十页的论文“通信的数学理论”中,虽然没有回答“信息是什么”,却提出信息的度量问题,创造了计算“信息熵”H的著名数学表达式:

$$H = -\sum P_i \log P_i$$

“信息熵”实际是对随机变量的位置和顺次发生几率相乘再求总和的数学期望。

② “耗散结构理论”的创始人是1977年诺贝尔化学奖获得者,物理化学家伊里亚·普里戈金(Ilya Prigogine, 1917-2003)教授。

③ 参考资料: <http://wiki.mbalib.com/zh-tw/%E8%80%97%E6%95%A3%E7%BB%93%E6%9E%84%E7%90%86%E8%AE%BA>

志和自我创造、演化的潜力，以及自我发展的路径，而不是简单地由外部条件影响、塑造。区块链伴随着算力的运行，会不断形成新的有序结构和产生新的能量，从这个角度说，区块链结构具有内生的演化逻辑。由此，需要提及“麦克斯韦妖”。<sup>①</sup>麦克斯韦做了一个实验：它能把一个封闭盒子里的本来热平衡的分子从一边“赶”到另一边，从而让一个热力学系统自动熵减，由此违反了熵增和熵最大原理。但是，这个实验已经被证明无法违背熵最大原理。因为实验中发现，“小妖”为了发现并分辨分子运动的快慢，以及消除自己的信息记录，本身也需要消耗能量，造成熵增。这个思想实验的意义之一在于，智能可以是耗散结构的方案。在某种意义上而言，区块链系统就是一种“麦克斯韦妖”，也就是说，区块链本身就是一种智能。

在这里应该指出，到目前为止，比特币及其区块链基础是符合耗散结构模型的，否则比特币也不足以维持十年之久。但是，按照比特币的初始设计，从现在到2140年的未来一百年间，比特币是否能够始终处于耗散结构，维持熵减，避免熵增的状态，受到太多的内生和外生因素的影响，难以做出简单的判断。但是，可以肯定的是：在过去四五十年间，传统法币在非耗散结构和熵增的过程中，发生了从算术级数到几何级数的转变，导致了全球货币体系走上了熵增的不归之路。

在结束序言前，我想告诉读者：邹均编写的本书，与其上一本著作《区块链技术指南》，以及市场上关于区块链的大量著作相比，以下几个方面值得注意。

其一，**宏观视野**。本书从当代互联网的局限和计算模式的演变的大背景出发，切入区块链的概念和技术框架，最后对区块链技术进行分解说明，使读者得以窥见区块链技术及其应用的全貌。

其二，**系统化**。本书对区块链技术相关的问题进行分解、梳理和归纳，归纳出区块链技术的“最小公约数”，最终形成了12个大主题，每个主题又细分成三个层次。这样的框架设计有益于系统讨论区块链技术。

其三，**科学的描述方式**。本书在讲解技术的过程中，超越日常生活的形象化举例，大量引用数学语言和符号语言进行描述，从而准确传达了概念和逻辑。

其四，**超越概念化**。本书没有停留在对概念的介绍上，而是大量讨论概念背后的实现过程、运行机制和交互关系，例如进行“关于比特币PoW能否解决拜占庭将军的问题”，“基于PoW的共识记账”等讨论。

其五，**重视和引用英语资料**。本书大量引用英语资料，一定程度上弥补了当下国内区块

---

① “麦克斯韦妖”，又被称为麦克斯韦精灵（Maxwell's demon），系1871年由英国物理学家麦克斯韦（James Clerk Maxwell, 1831-1879）为了说明违反热力学第二定理的可能性而设想的实验。

链文章和书籍的英文资料引用和研究普遍不足的缺憾。当然，可能因为团队分工与写作的限制，本书不乏重复和粗糙之处，距离精致化、凝练化还有一定距离。希望本书还有再版机会，得以完善。

可以预见，在区块链未来发展和成长过程中，还会出现一个又一个的“十字路口”，也就是一次又一次的挑战。但是，可以相信，区块链的内在学习能力将帮助其在“十字路口”做出正确选择。伴随区块链未来发展和成长，人类社会中的实体经济正在被观念经济所超越，与物理世界平行的数字世界全面崛起。区块链将是连接实体经济和观念经济，及与物理世界平行的数字世界的重要桥梁。这正是区块链真正的历史功能所在。

——朱嘉明，中国数字资产研究院院长、著名经济学家

（2018年6月3日初稿，2018年7月10日定稿）



## 序二 用发展的眼光看待区块链 *Foreword 2*

区块链是当今金融科技领域广泛探讨的热门话题之一。与传统技术对比，区块链具有难以篡改、安全性高、可靠性强、智能合约自动执行、网状协作更加透明等特征。在近年的落地应用中，区块链技术很好地支撑了比特币系统的正常运行。随着研究的不断深入，区块链在不同场景的应用研究蓬勃发展，业界对区块链技术的局限性有了更深刻的认识。具体主要有以下三个方面。

一是区块链的性能问题有待突破。目前来看区块链的性能问题主要体现在共识的效率，这是系统设计时面临的最大挑战。智能合约运行结果需要串行排序并在每个记账节点中形成共识，因此无法并行，这会制约节点的处理能力。

二是在隐私保护方面有待加强。传统上数据是保存在数据中心的，但是现在每个参与方都能够获得完整的数据备份，这也意味着数据是公开透明的。对于比特币而言，这样的解决方案也许够用，但如果想要区块链承载更多的业务，比如实名资产，这些信息如何保存在区块链上，验证节点在不知道具体信息的情况下如何执行合同。在这种情况下，就需要像零知识证明这种新型密码学方案来解决，这样也可以规避一些隐私泄露等问题。

三是升级修复机制有待探索。在公有链中，因为节点数量庞大，参与者身份不明，不可能关闭系统。因此，由于公有链不能关停，其错误修复也异常棘手，一旦出现问题，尤其是出现安全问题时将非常致命。

互联网近年来的迅猛发展及其与物理世界的深度耦合与强力反馈，已经根本性地改变了现代社会的生产、生活与管理决策模式。可以预见的是，未来在中心化和去中心化这两个极点之间，将会存在一个新的领域，各种区块链系统都会拥有不同的非中心化程度，以满足不同场景的特定需求。如何探寻到其中的最优均衡点，以及如何让各个区块链系统之间协作起来，则成为当前区块链研究和应用的关键。

总体来说，区块链技术有优点，也有缺点，仍在不断发展演进中。目前看，区块链若要实现真正落地，能够支撑实际业务，在技术层面仍需大量改进工作。对于监管者而言，面对不断演进的区块链技术，需要同步考虑相应的法律法规和技术标准，以加强监管，防范风险。

习近平总书记指出：“进入 21 世纪以来，全球科技创新进入空前密集活跃的时期，新一轮科技革命和产业变革正在重构全球创新版图、重塑全球经济结构。以人工智能、量子信息、移动通信、物联网、区块链为代表的新一代信息技术加速突破应用”<sup>①</sup>。区块链是一种可能成为未来金融基础设施的新兴技术，对其进行深入研究是我国金融科技工作的应有之义，亦是我国在核心技术上下工夫，不受制于人的关键之举。

本书作者们本着求真务实的态度，力求给读者提供一本完整、翔实地反映当前区块链技术体系的书籍，是广大区块链爱好者和从业者可以参考学习的工具书，值得推荐。

——姚前，中国人民银行数字货币研究所所长

---

<sup>①</sup> 节选自“中国科学院第十九次院士大会、中国工程院第十四次院士大会”上的讲话。

### 序三 区块链与系统安全 *Foreword 3*

作为一个新兴的融合型技术，区块链日益受到社会各界的关注。如何客观、正确地认识区块链，对待区块链是值得我们思考的第一个问题。我个人认为不能把区块链仅仅看成是一项技术，而更重要的是要把区块链看成是一种伟大的思想和方法，我们要运用区块链的思想和方法解决实际的问题，而不仅仅是坐而论道。第二个问题是我们应该在什么场景下应用区块链？我认为凡是在高价值的数据的管理、流通与共享领域都可以应用区块链。第三个问题是我们应该如何处理在区块链落地时所碰到的问题。区块链作为一种新兴事物，在实际落地中可能会出现各种问题。我们首先应该积极支持区块链的创新，对新出现的问题，及时分析研究，找出对策，确保区块链这个新兴事物能健康发展。

作为一个从事安全研究工作几十年的科研工作者，我认为，区块链应用落地目前碰到的最大问题将是安全问题。很多人可能不理解，认为区块链的核心技术之一是密码学，因此区块链应该是一种安全技术，怎么安全问题反而是区块链的最大问题呢？让我们来看看近期发生的安全事件：2018年6月，韩国加密货币交易所Coinrail遭到网络入侵，导致事件发生的24小时内：比特币下跌1000美元，以太币、瑞波币、比特币现金均现两位数跌幅，热门的EOS跌幅更是将近20%。加上过去很多由于安全漏洞造成区块链上数字资产的重大损失案例，大家就不难理解，区块链的安全问题为什么在整个区块链应用落地中是如此重要的问题了。

那么区块链项目现在面临了哪些主要的安全问题？答案是私钥的生成与保护、共识过程的中心化、智能合约代码漏洞、量子计算机威胁等。这些例子包括：2016年6月17日，黑客偷取众筹超过1.5亿美元的The DAO项目1/3的以太币，导致项目失败；2016年8月2日，因为多重签名漏洞，香港的比特币交易所时值约7000万美元的比特币被盗；2016年8月，Krypton受到51%攻击，导致Bittrex钱包中共21465个KR（代币）被盗；另外2018年4月



22 日, BEC( 代币 ) 由于代码溢出问题从 60 亿市值直接归零。

我们看到区块链系统面临的安全威胁是实在的, 并不是危言耸听。那如此重要的安全性应该怎么保障呢? 我们知道, 任何信息系统的安全包括保密性 ( confidentiality )、完整性 ( integrity ) 和可用性 ( availability ), 也就是常说的 ( CIA )。

从保密性上来说, 区块链系统以公钥的变形作为交易地址, 为用户提供了保密性; 零知识证明、环签名等, 为交易提供了保密性。区块链系统也大量应用了密码的前沿技术。

从完整性上来说, 签名验证算法保证了交易的完整性, 链式结构保证了数据的完整性, 高度同构结构, 保证了系统的完整性。

从可用性上来说, 每个副本节点保存相同的数据, 保证了数据的可用性, 多方共识机制保证了交易信息的可用性。

所以我们看到区块链系统相对传统的系统来说, 从三个方面提供了比较强的安全属性。

但是我们要清醒地认识到, 任何系统的安全取决于系统安全的最薄弱环节, 也就是常说的木桶的容量不取决于最长板, 而是取决于最短板。

先从区块链系统的安全性方面来看一下算法的安全性。目前的区块链用的加密算法基本上是安全的, 但是 Shor 量子算法在理论上能破解在区块链上广泛使用的公钥系统。如果量子计算机实现商用, 那么现在区块链上的公钥系统算法都会变得不安全, 所以说, 区块链的密码学算法的安全性是相对的。

接下来看看区块链协议的安全性。① 区块链协议本身可能就有逻辑缺陷, 像受到黑客攻击的区块链系统共识机制, 其实也是利用协议上存在的安全问题。② 所有的数字货币系统安全性高度依赖私钥, 导致私钥安全与否对整个区块链系统影响非常大。③ 区块链系统实现安全性存在隐患。2016 年 10 月, 国家互联网应急中心选取了 25 款具有代表性的区块链软件, 发现了大量安全隐患, 所以目前区块链的软件从代码的安全性来说, 是非常令人担忧的。

那么我们面对这些安全问题应该怎么办? ① 针对算法安全性, 应对措施之一就是采用抗量子的算法, 采用聚合签名策略、环签名策略、门限签名策略, 总之要采用新的密码技术, 这些密码技术本身要经得起考验。② 针对协议安全性, 需要在 PoW 中使用防 ASIC 杂凑函数, 使用更有效的共识算法和策略。③ 针对实现安全性的应对方法, 需要对关键代码进行严格完整测试, 以及采用更加安全的智能合约, 对智能合约进行形式化验证。

除区块链本身安全外, 使用安全性也是安全重要一环, 主要是对私钥存储、使用这方面进行保护。特别对交易所来说, 因为交易所聚集了大量的数字货币, 成为黑客或者一些敌对、恐怖组织针对的重要目标, 所以选择安全交易所非常重要。

在建设网络强国的指导思想下, 我国大力发展区块链、大数据技术, 既要注重经济效益,

也要注重安全问题，二者缺一不可！区块链系统安全十分重要，目前区块链系统安全面临着巨大的挑战，所以需要我们积极应对。

近年来区块链技术吸引了大量金融和科技企业进行投资，许多投资者认为这项技术具备改变多个行业的能力（如医疗、公共事务、能源、工业、金融行业等），但现状是很多人仍然不了解这项技术及其成熟度，对区块链抱有不切实际的幻想。他们希望部署区块链来获取利益，但对区块链的核心能力了解并不充分。这本书给读者一个全面系统理解区块链的机会，既没有夸大，也没有贬低，观点比较客观中肯，值得推荐。

——斯雪明，复旦大学教授、中国计算机协会区块链专委会主任

## 为什么要写这本书

2017年8月1日，在比特大陆（Bitmain）的大力推动下，比特币社区发生了比特币现金（BCH）分叉事件。比特币一分为二，BCH改变了中本聪原先制定的规则，将区块大小从1MB扩容到了8MB。后续比特币社区陆续有20多个分叉项目出现。与此同时，ICO项目在国内层出不穷，各种空气币、资金盘等借区块链圈钱诈骗的事件不断出现。2017年9月4日，中国人民银行等七部委，宣布ICO非法，关停国内的场内虚拟货币交易所，并责令各类ICO项目清退募集的虚拟货币。可以说，BCH分叉事件标志着区块链行业的发展从“春秋时代”过渡到“战国时代”。

另一方面，透过币圈纷繁喧嚣的现象，我们也要看到，近年来区块链技术发展进入一个百舸争流、百家争鸣的时代。以比特币为代表的区块链1.0和以以太坊为代表的区块链2.0虽然在数字资产发行方面取得了很大成功，但在区块链应用落地方面却非其所长。业界涌现出很多项目，都号称是区块链3.0的代表，争相提出新的共识机制、分片机制、数据结构、跨链协议、链下计算、状态通道、隐私保护算法、治理机制和安全措施等新技术。新的理念，像区块链操作系统、区块链中间件、区块链网络、分布式Oracle层出不穷。自笔者2016年出版《区块链技术指南》一书以来，区块链技术和应用已经有很大的发展，业内需要有一本更新的，更加全面、翔实地反映当前区块链技术现状的技术参考书。

最早酝酿这本书是在2017年的5月中旬，由北邮在线董事长于斌和主任张权召集了几个区块链专家，讨论联合出书的可行性。自那以后，参与写作的各位作者开始了为期一年的漫长业余写作。其中所遇到的困难远远超出了当初的想象。也经常有朋友问，为什么要在区块链正值风口的时候，选择写书这种既缓慢又没有多少收益的事情，特别是笔者“重操旧业”，再作冯妇，又花一年时间来写第二本区块链技术的书呢？细想原因有三：一是兴趣使然；二





是两年来来自读者的鼓励和鞭策使我不能停下脚步；三是世界上总得有人做苦差，不可能每个人都可以分得美差。每每想到此处，心中也就释然。

在利用业余时间写作的这一年，与其说是写作区块链图书的过程，不如说是一个不断学习和思考区块链，不断质疑、校正、改变自己区块链思想的过程。区块链在这个科技飞速发展、社会热点风云变幻的时代，能持续得到这么广泛和持久的关注，原因在于其独特的技术属性产生的魅力。它的点到点对等网络、去中心化的设计、信息防伪防篡改的保障、基于算法的共识机制，由此衍生出的价值传递能力，按规则客观、忠实、自动履行合约的能力，隐私保护的能力，透明、历史可追溯的特点，使它不仅可以像传统技术那样只是提升生产效率，更重要的是可以改变生产关系。

因此社会很多不同阶层、不同行业、不同群体的人士都很大程度上对区块链感兴趣，原因是从区块链中看到了他们的理想工具。监管机构和执法机构看到了区块链加强监管、追踪证据的可能性；技术社区看到了技术创新的一个新方向；IT 从业人员看到职业发展的新方向；创业者看到了创业的新领域；投资者看到了致富的新捷径；知识产权保护者看到了区块链确权、存证、维权的前景；反数据垄断者看到了区块链的防止大平台垄断的希望；制度设计者和管理者看到了降低信任成本从而降低交易成本的潜力；改革者看到了用区块链来革除弊端，并在数字经济时代重构组织架构、重塑生产关系的前景。

所有这些扮演不同角色，怀揣不同目的和想法的群体都不约而同地关注区块链，因此可以想象，区块链领域是多么热闹。这里面会有真知、洞见；也有炒作、夸大、神化；甚至有歪曲、抹黑。没有先入为主的广大读者群体，希望看到的是一本还原区块链技术本质，不加修饰、点缀，甚至不掺杂主观思想的纯技术书籍。这也是笔者在写作过程中才发现的一个写作驱动点。

但从另一方面来看，每个区块链的观察者或从业者，都会有自己角度的观察和解读，甚至提炼成的观点和思想。这些观点和思想需要讨论，需要碰撞，需要形成共识。因此，笔者也愿借本书前言一隅，谈谈自己对区块链的看法。

笔者认为，从实质上看，区块链是一个带有共识机制的分布式计算机网络，而共识的结果形成一个不可篡改的档案库。共识机制是一套协议，也是一套规则。在这个分布式的计算机网络中，大部分节点都需要遵守这套规则。如果不是这样，这个区块链系统就不能正常工作。要保证系统能正常工作，共识机制的设计就很重要。它必须能激励参与节点遵守规则，同时最好能惩罚不遵守规则的节点。需要共识的是每个节点中在某个时间点的状态，也就是说形成公认的、确定性的状态。在这里，可以把节点抽象成一个计算机科学里的概念——有限状态机（英文为 Finite State Machine 或 Automata）。有限状态机指的是响应外界特定触发条



件，并按一定规则做状态转换的抽象机器。例如我们日常所见的自动柜员机（ATM）就是一个有限状态机。它根据输入的指令，使系统内部发生相应的状态变化。以取钱为例，从等待输入到减少库存钞票、输出钞票，再回到等待输入等一系列状态变化。在ATM的这个例子中，系统变化的规则是由后台中心化的银行核心系统决定的。而在区块链环境中，可以想象成每个ATM机都是独立的，有自己的账本系统，不受银行的核心系统控制。在任何一个ATM机上取钱，其他的ATM机都要进行验证，最后大部分ATM机要达成对账本状态的一致性确认，这样才能把钱取走。

也就是说，区块链系统是一个在分布式系统中不通过中心管控节点而能使各节点保持步调一致的系统。从这个意义上来看，区块链是带有颠覆性的创新。过去，只有通过中心化的管控，才能使分布式的系统步调一致。而现在，通过运行在各节点的共识算法，就能使得分布式的系统按规则形成“自治”。

我们知道，自然界的系统都有一个自发的、逐渐变得无序的趋势，例如一间房，在没人打扫的情况下，会逐渐变得尘埃遍布。这就是热力学第二定律说的：一个封闭的系统总是向熵增加的方向发展。所谓熵，在热力学中就是一个无序、不确定性的量度。在香农创立的信息论中，信息是确定性的一个度量，而熵是信息量的一个度量。熵越大，越无序，信息量越少。而从这个意义上来说，区块链系统是一个熵值减少的系统，因为共识所确认的状态就是信息，而信息也就是有序和确定性。

一个运行良好的区块链系统，给我们提供了一个低成本的降低熵值的系统。毫无疑问，这会给关注社会、组织管理的人带来很大的启发意义。如何应用区块链来变革传统的制度、组织管理方式就成为一个需要重点关注的问题。不当的推广与不合时宜的应用，反而会阻碍区块链的应用和发展。

非常具有讽刺意味的是，区块链是一种依托算法建立信任的技术，但恰恰有很多人在区块链领域有非常多的不诚信做法。他们或许不知道，他们的很多言行，早已记录在大部分关注区块链的人们心中的账本上，就像区块链不可篡改和可追溯那样，历史不会遗忘那些借炒作区块链来达到损人利己目的的人。

区块链领域的从业者，需要建立行业自律，急需形成共识，建立一个区块链行业文化，以体现区块链的透明、诚信、公正、公平的特点。

因此，写这本书的目的，不仅希望从技术层面提供一本接近客观、真实的区块链书籍，更希望在求真务实的基础上，使广大关注区块链的社区能凝聚共识，弘扬区块链领域的正能量，建立行业自律，使得区块链技术和应用得到健康持续发展。





## 本书特色

希望本书能为想要系统了解区块链技术的从业人员提供比较完整、及时、翔实的指导。本书首先对基本概念做了详细讲解。然后就一些区块链基础知识，如区块链架构、密码学、共识算法、P2P 网络等做了详细阐述。再结合区块链的主流平台，像比特币、以太坊、超级账本 Fabric 来讲解不同区块链平台实现的原理。接着分析向区块链 3.0 发展的不同新项目的特点，其中重点讲述当前具有区块链操作系统特性的 EOS 项目的技术现状。同时也简单描述代表区块链 3.0 的不同发展方向中的项目，包括：区块链网络中的闪电网络、Cosmos、Polkadot；侧链中的 Rootstock、Lisk；DAG 中的 IOTA、Byteball、HashGraph；新型区块链系统 Cardano；区块链存储或存储挖矿系统 StorJ、Burstcoin、Filecoin 等。最后讨论区块链领域存在的问题以及区块链测评等问题。

技术的全面性、翔实性、实操性和技术更新的及时性应该是本书的最大特色。其中，书中特别强调实操，在很多章节都有可供读者自己动手配置环境、运行代码的参考性内容。要真正掌握一门新技术不容易，特别是像区块链这种融合多种技术的新型技术架构。我们学习区块链，要做到“知行合一”，不单是要读区块链书籍，还要亲自动手配置环境和编写代码，以此来验证学到的知识。更重要的是，还要把学到的区块链知识应用于实际。只有这样，才是真正的“知”。

## 读者对象

- ☐ 区块链应用开发人员
- ☐ 区块链架构师
- ☐ 区块链底层开发人员
- ☐ 计算机专业的本科或研究生
- ☐ 区块链相关的业务和管理人员
- ☐ 其他对区块链技术感兴趣的人员

## 如何阅读本书

本书分为三篇，共计 12 章内容。

第一篇为**核心技术篇**，着重讲解区块链相关的基本概念、基础架构和核心技术，包括以下 7 章内容。





第1章 从互联网发展碰到的局限和挑战、分布式计算的演变出发，引出比特币的诞生背景，并简单介绍比特币的特点和局限。最后重点阐述区块链相比于互联网在信任建立和价值传递方面的重大意义。

第2章 介绍了区块链的基本概念和区块链的基本技术，覆盖 P2P 网络、密码学、共识机制、智能合约等各核心技术的基本概念。

第3章 给读者展现了一个整体的区块链架构。通过对不同区块链异同点的分析，总结区块链的本质，进而从本质和发展角度给出了区块链的架构模型和部署模型。

第4章 系统介绍了区块链常用的密码学技术原理，包括哈希算法、公用系统的非对称加密原理、数字签名技术和零知识证明、环签名等隐私保护算法。

第5章 系统介绍了强一致的共识算法和最终一致性共识算法，并讨论了主要区块链平台的共识算法。

第6章 介绍了区块链 P2P 网络协议的定义、产生、特征等基本内容，详细阐述了区块链 P2P 技术的基本结构、工作过程、网络构架等内容，最后重点以比特币和以太坊为例，介绍其 P2P 协议的基本内容和工作原理。

第二篇为实战篇，着重讲解主流区块链平台的架构原理与应用，包括以下3章内容。

第7章 详细深入地介绍了比特币系统的基本部分，包括区块数据存储、共识机制、密码算法、脚本引擎以及 P2P 网络处理模块，并讲解了比特币如何通过组合这些技术实现一个点对点的电子现金系统。

第8章 介绍了以太坊的出现背景、关键概念，详细说明了以太坊的核心架构。同时对以太坊智能合约和流行的高级合约语言 Solidity 也做了详细说明，构建了一个以太坊测试网络，并在网络上编译、部署和运行智能合约的案例。

第9章 主要阐述了超级账本权限链 Fabric 项目的架构原理和应用开发流程，并用实例讲解如何使用 Fabric 开发应用。

第三篇为进阶篇，讲解区块链的发展方向、潜力框架、常见问题，以及测评方法，包括以下3章内容。

第10章 总结了区块链技术发展的主要方向，简单介绍了主流平台以外一些富有特色的区块链平台以及它们的特点和应用，包括 EOS、Cardano、IOTA 等。

第11章 介绍了许多区块链领域内的常见问题，包括区块链的技术局限、数据冗余、安全性、各种共识协议的弱点、交易速度、51% 攻击问题、女巫攻击、交易所及以太坊智能合约安全漏洞，并讨论了相应的应对措施。

第12章 从区块链系统整体出发，采用正交分析法，从6个层面和8大类质量指标来设



计评测点和测试用例。

如果你没有充足的时间完成整本书的阅读，可以选择性地进行重点章节的阅读。如果你是区块链初学者，可以重点学习第一篇。如果你是一位有着一定经验的资深人员，本书可能会是一本可提供参考的案前书，你可以直接跳到第二、三篇读你所感兴趣的章节，如比特币、以太坊或超级账本 Fabric 等章节。

## 勘误和支持

由于笔者的水平有限，加之编写时间仓促，书中难免会出现一些错误或者不准确的地方，恳请读者批评指正。如果你有更多的宝贵意见或希望和笔者沟通，可以采取以下方式。

1) 加入微信群“区块链核心技术读者交流群”，添加微信号 zhixinglian23 或 liandajun3986 入群。

2) 关注微信公众号“链信 Chain2Trust”。

期待能够得到你们的真挚反馈，让我们在技术之路上互勉共进。

## 致谢

这本书是集体智慧的结晶。首先感谢参与策划和写作的伙伴们。他们是北邮在线董事长于斌博士、IBM 架构师庄鹏先生、清华大学信息学院邢春晓院长、赣州新链总经理范金刚先生、VMware 应用开发首席架构师张海宁先生、《白话区块链》第一作者蒋勇先生、亚投行 IT 战略顾问邵周先生、广州大学唐屹教授、IBM 架构师郭莹城先生、联动优势首席架构师刘胜先生、清华大学张桂刚博士、VMware 程序员陈家豪。他们花了很多心血，牺牲了大量的业余时间来收集资料、写作，贡献他们各自的章节，体现了他们的专业精神。特别是庄鹏先生，不单提供了全面、翔实的内容，还协助笔者做审校。另外要鸣谢联动优势的向峥嵘博士，他协助笔者做本书的审校、统稿，为本书的顺利出版做了大量工作。同时需要感谢北京大学密码学研究员陈宇教授，美国 Visa 研究员、波士顿大学密码学博士陈一镭，他们对密码学章节进行审稿并提供了宝贵意见。另外要感谢本书的特约策划、北邮在线的张权主任，他不断地跟进本书进度，使笔者不敢有些许松懈。

笔者衷心感谢为本书作序的各位前辈和大师。其中中国数字资产研究院朱嘉明院长在笔者写书过程中一直给予很多鼓励，并为本书初稿提供了宝贵的意见。他另外用近一个月的时间为本书作了一个长达万言，极具思想深度和原创观点的序言。序言中涉及不同领域的专业





知识，朱教授都反复和各领域专家进行讨论、充分论证，力求做到准确无误。朱教授的严谨治学态度堪称学术界的楷模。中国人民银行数字货币研究所姚前所长在百忙之中抽出时间为本书作序，指出需要用发展的眼光来看待区块链和区块链技术；中国区块链基础技术与应用协同创新联盟理事长、中国计算机协会区块链专委会主任、复旦大学斯雪明教授也在繁忙的教学、科研工作中抽出时间为本书作序。大师们严谨治学、乐于帮助后进的精神值得我们学习。

感谢区块链项目的官网文档和开源社区，在写作期间提供给我们准确的参考材料。

感谢中国可信计算创立者、中国工程院沈昌祥院士对本书提出了宝贵的建设性批评意见；也感谢中国大数据技术与应用联盟副理事长赵平生先生、狗狗币中国区社区创始人周朝晖先生、中国数字资产研究院秘书长林茜女士、冯永强律师、OFBank 创始人刘大鸿先生、PPKPub 创始人陈晖先生、零壹财经 CEO 柏亮先生、零壹财经编辑孙爽女士、青岛链湾区块链研究院执行院长闫祖德、中国移动通信联合会国家区块链创新应用联盟秘书长陈晓华教授、布比 CEO 蒋海博士、中科院百人计划贺海武教授、太一云总裁邓迪博士、太一云研究院首席研究员甘国亮博士、爱立示 CEO 谈健博士、OFBank 运营副总王飞先生为本书的成书提供了帮助和指导。另外要感谢中国数字资产研究院的张宇先生、中国数字资产研究院秘书王婉卿女士、中国数字资产研究院郑延伟先生、博彦科技的赵丹女士、知行链总经理曾健先生、知行链 CTO 盛义东先生和知行链市场总监张娜女士对出版该书的帮助。

最后感谢巴比特和 CSDN 社区的各位技术专家们的博客文章，每次阅读必有所获，有所启发。

最后要鸣谢中国数字资产研究院、中关村区块链产业联盟、中国电子学会区块链专委会、中国移动通信联合会、国家区块链创新应用联盟和中国计算机协会区块链专委会的大力支持！

## 特别致谢

最后，我要特别感谢我的太太 Annie、女儿 Beverley 和儿子 Skyler，我为写作这本书，牺牲了很多陪伴他们的时间，但也正因为有了他们的付出与支持，我才能坚持写下去。

同时，感谢我的父母和兄弟姐妹，虽然幼时家境艰难，但儿时良好的家教以及和睦互助的家庭环境使我具备了健康的心智，并受用无穷。

谨以此书献给我最亲爱的家人，以及众多热爱区块链技术的朋友们！

邹 均





# 目 录 Contents

作者简介

序一 处于“十字路口”的区块链技术及其应用

序二 用发展的眼光看待区块链

序三 区块链与系统安全

前言

## 第一篇 核心技术篇

### 第1章 互联网下半场的主角—— 区块链····· 2

1.1 当代互联网的局限····· 3

1.2 计算模式的演变····· 4

1.2.1 分布式计算的出现····· 4

1.2.2 分布式计算的挑战····· 5

1.2.3 比特币“突围”····· 6

1.2.4 比特币的局限····· 8

1.3 区块链的演化路径····· 9

1.3.1 区块链与互联网意义····· 9

1.3.2 区块链概念的提出····· 9

1.3.3 区块链的社会价值和意义····· 9

1.4 小结····· 12

### 第2章 区块链基本概念····· 13

2.1 区块链技术研究缘由····· 13

2.1.1 区块链用例描述：比特币····· 14

2.1.2 区块链需要研究哪些关键技术····· 17

2.2 区块链模型····· 18

2.2.1 数据区块····· 20

2.2.2 链式结构····· 21

2.2.3 Merkle 树····· 22

2.3 网络通信层关键技术····· 22

2.4 数据安全与隐私保护关键技术····· 26

2.5 共识层关键技术····· 32

2.6 区块链技术标准····· 33

2.7 小结····· 34

### 第3章 区块链架构····· 36

3.1 区块链架构和传统 IT 架构的  
异同····· 36

3.2 区块链架构模型····· 37

3.2.1 区块链系统的共性····· 37

3.2.2 区块链的差异性····· 40



|                              |    |                                      |     |
|------------------------------|----|--------------------------------------|-----|
| 3.3 区块链的参考模型 .....           | 42 | 第5章 共识算法 .....                       | 83  |
| 3.3.1 一个参考架构 .....           | 42 | 5.1 分布式共识算法背景 .....                  | 83  |
| 3.3.2 区块链高阶架构模型 .....        | 43 | 5.1.1 拜占庭将军问题 .....                  | 85  |
| 3.3.3 区块链跨链本质与架构模型 .....     | 52 | 5.1.2 共识系统的基本定义 .....                | 88  |
| 3.3.4 区块链部署模型 .....          | 56 | 5.1.3 Fisher-Lynch-Paterson 定理 ..... | 89  |
| 3.4 区块链的数据模型 .....           | 58 | 5.1.4 CAP 定理 .....                   | 91  |
| 3.5 区块链的能力模型 .....           | 62 | 5.2 强一致性非拜占庭共识算法 .....               | 93  |
| 3.6 小结 .....                 | 64 | 5.2.1 Viewstamped Replicaton .....   | 93  |
| 第4章 密码学 .....                | 65 | 5.2.2 Paxos 共识算法 .....               | 96  |
| 4.1 哈希算法 .....               | 66 | 5.2.3 其他类 Paxos 共识协议 .....           | 98  |
| 4.1.1 密码学哈希函数简介 .....        | 66 | 5.2.4 强一致性非拜占庭共识算法小结 .....           | 100 |
| 4.1.2 哈希函数的性质及应用 .....       | 67 | 5.3 强一致性拜占庭容错共识算法 .....              | 101 |
| 4.1.3 区块链中的哈希函数 .....        | 69 | 5.4 非强一致共识算法 PoW 机制 .....            | 104 |
| 4.2 Merkle 树 .....           | 70 | 5.5 PoS 机制 .....                     | 110 |
| 4.2.1 哈希指针 .....             | 70 | 5.5.1 点点币 PoS 机制 .....               | 111 |
| 4.2.2 Merkle 哈希树 .....       | 72 | 5.5.2 NXT PoS 机制 .....               | 112 |
| 4.3 公钥密码算法 .....             | 73 | 5.5.3 Tendermint PoS 机制 .....        | 113 |
| 4.3.1 密码算法简介 .....           | 73 | 5.5.4 Ethereum Casper PoS 机制 .....   | 116 |
| 4.3.2 公钥密码算法 .....           | 74 | 5.5.5 LPoS 机制 .....                  | 117 |
| 4.3.3 区块链中使用的椭圆曲线 .....      | 75 | 5.5.6 DPoS 机制 .....                  | 117 |
| 4.3.4 数字证书 .....             | 75 | 5.6 Ripple 共识算法 .....                | 118 |
| 4.4 数字签名 .....               | 76 | 5.7 小结 .....                         | 120 |
| 4.4.1 数字签名简介 .....           | 76 | 第6章 P2P网络 .....                      | 122 |
| 4.4.2 数字签名标准与 ECDSA 算法 ..... | 77 | 6.1 P2P 网络简介 .....                   | 122 |
| 4.4.3 其他的数字签名方法 .....        | 78 | 6.2 P2P 网络核心数据结构与算法 .....            | 127 |
| 4.5 零知识证明 .....              | 79 | 6.2.1 P2P 网络数据结构与算法 .....            | 127 |
| 4.6 区块链中的隐私问题 .....          | 80 | 6.2.2 主流数据结构 DHT 与算法 .....           | 128 |
| 4.7 小结 .....                 | 82 | 6.2.3 区块链 P2P 网络协议 .....             | 136 |
|                              |    | 6.3 小结 .....                         | 147 |



## 第二篇 实战篇

### 第7章 比特币 ..... 150

#### 7.1 比特币的特点 ..... 150

#### 7.2 比特币的 P2P 网络 ..... 153

##### 7.2.1 点对点的钱包节点分类 ..... 153

##### 7.2.2 全节点的分布式存储 ..... 154

##### 7.2.3 交易和区块在节点间的传播 同步 ..... 156

#### 7.3 比特币的发行机制 ..... 157

##### 7.3.1 总量上限 2100 万的实现 ..... 157

##### 7.3.2 打包区块的原理 ..... 158

##### 7.3.3 矿池与矿工的关系 ..... 161

#### 7.4 比特币的账号系统 ..... 163

##### 7.4.1 私钥与公钥 ..... 163

##### 7.4.2 签名信息与校验签名 ..... 165

##### 7.4.3 脑口令 ..... 165

##### 7.4.4 荣耀地址与批量地址 ..... 167

##### 7.4.5 多重签名地址 ..... 168

##### 7.4.6 隔离验证 SW 地址 ..... 177

#### 7.5 比特币的生态系统 ..... 177

#### 7.6 开发实施一个比特币存证应用 ..... 179

##### 7.6.1 环境准备 ..... 179

##### 7.6.2 示例程序 ..... 182

#### 7.7 小结 ..... 184

### 第8章 以太坊 ..... 185

#### 8.1 以太坊关键概念 ..... 186

#### 8.2 以太坊的架构 ..... 188

##### 8.2.1 以太坊数据模型 ..... 189

##### 8.2.2 以太坊的应用架构 ..... 195

#### 8.3 以太坊智能合约 ..... 197

##### 8.3.1 合约类型和调用示例 ..... 197

##### 8.3.2 合约编译和部署过程 ..... 199

##### 8.3.3 Solidity 高级合约语言 ..... 201

##### 8.3.4 案例：构建、编译与部署一个 智能合约 ..... 203

#### 8.4 以太坊适用场景剖析 ..... 216

#### 8.5 小结 ..... 217

### 第9章 超级账本Fabric ..... 218

#### 9.1 Fabric 基础架构 ..... 219

##### 9.1.1 架构概述 ..... 219

##### 9.1.2 主要组件 ..... 220

##### 9.1.3 P2P 网络 ..... 221

##### 9.1.4 通道 ..... 222

##### 9.1.5 分布式账本 ..... 222

##### 9.1.6 共识机制 ..... 223

##### 9.1.7 智能合约 (链码) ..... 224

##### 9.1.8 成员服务提供者 ..... 225

##### 9.1.9 交易流程 ..... 225

#### 9.2 架构详细原理 ..... 227

##### 9.2.1 成员身份管理 ..... 227

##### 9.2.2 通道的结构 ..... 232

##### 9.2.3 链码 ..... 238

#### 9.3 应用开发流程 ..... 245

##### 9.3.1 前期准备 ..... 246

##### 9.3.2 定义 Fabric 集群 ..... 246

##### 9.3.3 启动 Fabric 集群 ..... 248

##### 9.3.4 链码设计 ..... 248

##### 9.3.5 链码部署 ..... 251

##### 9.3.6 SDK 简介 ..... 251



|        |            |     |
|--------|------------|-----|
| 9.3.7  | 应用的 API 开发 | 254 |
| 9.3.8  | 界面开发       | 257 |
| 9.3.9  | 集成         | 260 |
| 9.3.10 | 测试应用       | 261 |
| 9.3.11 | 扩展应用中的组织数目 | 263 |
| 9.4    | 小结         | 271 |

### 第三篇 进阶篇

## 第10章 其他区块链平台 274

|        |                                  |     |
|--------|----------------------------------|-----|
| 10.1   | 区块链架构存在的问题和挑战                    | 274 |
| 10.2   | 区块链平台的典型需求和发展方向                  | 276 |
| 10.2.1 | 区块链平台的典型需求                       | 276 |
| 10.2.2 | 区块链平台的发展方向                       | 277 |
| 10.3   | 其他区块链平台                          | 278 |
| 10.3.1 | 以太坊：区块链龙头的转型升级                   | 278 |
| 10.3.2 | EOS：区块链操作系统                      | 281 |
| 10.3.3 | Cardano：具有隐私和法规的区块链              | 289 |
| 10.3.4 | 基于区块链的支付协议：Ripple 与 Stellar      | 291 |
| 10.3.5 | 侧链代表：RootStock、Polkadot 和 Cosmos | 294 |
| 10.3.6 | 分片扩容：Zilliqa 叫板 Visa             | 296 |
| 10.3.7 | 跨链技术：价值互联网的纽带                    | 297 |
| 10.3.8 | DAG：区块链的革新                       | 300 |

|         |                    |     |
|---------|--------------------|-----|
| 10.3.9  | Hashgraph：区块链的新竞争者 | 304 |
| 10.3.10 | 区块链存储              | 306 |
| 10.3.11 | 安全和隐私保护            | 310 |
| 10.4    | 一句话解释主要加密货币        | 310 |
| 10.5    | 小结                 | 312 |

## 第11章 区块链常见问题剖析 313

|        |                |     |
|--------|----------------|-----|
| 11.1   | 区块链的技术局限       | 313 |
| 11.1.1 | 区块链不可能三角       | 313 |
| 11.1.2 | 数据冗余           | 313 |
| 11.1.3 | 区块链安全性         | 314 |
| 11.1.4 | 挖矿和其他共识协议的弱点   | 315 |
| 11.1.5 | 交易速度           | 316 |
| 11.2   | 区块链的安全问题       | 317 |
| 11.2.1 | 51% 攻击问题       | 317 |
| 11.2.2 | 女巫攻击           | 317 |
| 11.2.3 | 交易所            | 318 |
| 11.2.4 | 以太坊智能合约安全漏洞    | 320 |
| 11.2.5 | 区块链安全性的测试指标    | 321 |
| 11.3   | 挖矿和共识协议的弱点     | 322 |
| 11.3.1 | 中本聪一失之虑        | 322 |
| 11.3.2 | 挖矿和算力集中困境      | 322 |
| 11.3.3 | 其他共识算法及其问题     | 323 |
| 11.4   | 交易效率问题         | 323 |
| 11.4.1 | 比特币和以太坊的交易效率困境 | 323 |
| 11.4.2 | 比特币扩容          | 324 |

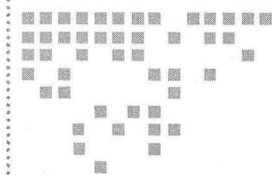
|                               |            |                  |            |
|-------------------------------|------------|------------------|------------|
| 11.4.3 比特币的隔离验证、闪电网络与侧链 ..... | 324        | 12.2 评测的策略 ..... | 333        |
| 11.4.4 基于 DAG 的提速技术 .....     | 326        | 12.3 应用层评测 ..... | 337        |
| 11.4.5 其他提速思路 .....           | 328        | 12.4 合约层评测 ..... | 338        |
| 11.5 系统升级维护问题 .....           | 328        | 12.5 激励层评测 ..... | 340        |
| 11.5.1 硬分叉史记 .....            | 328        | 12.6 共识层评测 ..... | 341        |
| 11.5.2 系统升级维护难题和分叉 .....      | 329        | 12.7 网络层评测 ..... | 345        |
| 11.6 小结 .....                 | 329        | 12.8 数据层评测 ..... | 345        |
| <b>第12章 区块链评测 .....</b>       | <b>330</b> | 12.9 辅助工具 .....  | 347        |
| 12.1 评测的难点 .....              | 331        | 12.10 小结 .....   | 350        |
|                               |            | <b>后记 .....</b>  | <b>351</b> |

## 第一篇 *Part 1*

# 核心技术篇

- 第1章 互联网下半场的主角——区块链
- 第2章 区块链基本概念
- 第3章 区块链架构
- 第4章 密码学
- 第5章 共识算法
- 第6章 P2P 网络





## Chapter 1 第 1 章

# 互联网下半场的主角——区块链

作者：范金刚

隔木对小时

马克思说过一句名言：“蒸汽、电力和自动纺织机甚至是比巴尔贝斯、拉斯拜尔和布朗基（19世纪法国著名的革命家）更危险万分的革命家。”马克思的意思非常明显：生产力的革命是一切生产关系革命的基础。

自蒸汽动力工业革命以来，人类社会生产力的发展进入了新纪元，从线性发展跃迁到指数级发展。尤其是进入互联网时代以来，重大技术突破层出不穷，技术进步的速度已经远远超过了人类变革生产关系以适应新的生产力的速度。科学技术的发展一方面带来了经济的增长和社会的进步，另一方面也带来了安全、伦理方面的挑战。当今互联网的发展，特别是在中国已走完了由人口红利驱动的上半场，目前已经到了一个十字路口。一方面，经过20多年的发展，互联网已经改变了人们生活和工作方方面面，带来了工作效率的大幅提升和生活的极大便利。但另一方面，由于互联网依靠中心化机构建立信任，使得互联网日趋中心化，由此带来的数据垄断、隐私侵犯、权益分配不公等问题也成为制约其发展的瓶颈。因此，寻找互联网下半场的主角成了当今社会的呼声。

“区块链”（Blockchain）这一概念起源于2009年初诞生的比特币。其创新的分布式去中介化的信任体系改变了传统互联网依托中心化机构建立和维护信任的机制，它能够帮助人类在数字经济时代变革社会生产关系，跟上以科技为主导的生产力发展的步伐，给金融、经济、科技甚至政治等各领域带来深远的影响。近年来，区块链受到了国内外的高度关注。2018年达沃斯论坛上，来自全球的顶级区块链企业领袖认为，区块链技术是人类历史上的第四次工业革命。区块链也被期望在互联网下半场成为驱动互联网发展的主角。

## 1.1 当代互联网的局限

随着计算机科学技术的飞速发展，互联网已经成为信息社会发展的重要保证，但也引发了各种各样的隐患，如网络安全漏洞、黑客攻击威胁、隐私信息泄露、数据造假、版权侵犯、电信诈骗以及网络治理难以统筹等问题，限制着互联网的进一步发展。

### 1. 网络信息安全

近年来，国内外网站数据和个人信息泄露事件频发，对政治、经济、社会的影响逐步加深。黑客攻击、网上盗窃、网上欺诈、网络病毒等网络安全问题层出不穷，对社会危害极大；数据权益侵犯、个人隐私数据滥用问题屡见不鲜，“衍生灾害”严重。由于互联网传统边界的消失，各种数据遍布终端、网络、手机和云上，加上互联网黑色产业链的利益驱动，数据泄露威胁日益加剧，甚至个人生命安全也因此受到威胁。

在国外，美国大选候选人希拉里的邮件泄露事件直接影响到美国大选的进程；雅虎两次账户信息泄露涉及约 15 亿的个人账户被曝光后，致使美国电信运营商威瑞森 48 亿美元收购雅虎计划搁置甚至取消。2018 年 3 月，媒体曝出英国大数据分析公司“剑桥分析”，利用非法手段截取 Facebook 上 8700 万用户信息，继而接受政治团体的资金，帮助他们通过大数据分析和媒体手段来影响美国大选以及英国脱欧的投票结果。

### 2. 数据版权

互联网作为一种新型的传播媒体，已成为版权产业的一个重要组成部分。由于互联网产品千姿百态，信息传播的速度和密度也随之倍增，从而可能招致的后果便是知识产权纠纷案例呈几何式增长。比如在微信领域，《腾讯知识产权保护白皮书》称：自 2015 年 1 月至 2016 年 12 月，微信共收到针对个人账号的侵权投诉 10.7 万余件，其中，知识产权侵权投诉主要涉及商标权、著作权、专利权等。同一时期，微信共收到针对公众号文章侵权投诉 6.1 万余件，其中著作权侵权投诉近一半，占总投诉量的 41%。

此外，由于部分网络用户对互联网“共享”精神误解为“凡是在网络上搜到的就能无偿使用”，因此出现了大量使用他人作品的侵权行为。许多网站采用网络爬虫技术自主抓取网络内容，缺乏对他人版权作品的甄别，也引发了大量侵权案例。甚至出现一些互联网媒体公司通过爬虫搜索和大数据分析做成独角兽公司的情况。

在我国，著作权受现行法律保护，但在实际纠纷发生时，权利人通常难以举证内容创作时间以证明权利归属。在这种情况下，国内知识产权意识淡薄，很多人不懂得如何保护自己的著作版权，而很多人不经意间就侵犯了他人的版权。因此，国内互联网上的数据权益保护还处于比较混乱的状态。

### 3. 新的挑战

第一代互联网的起点是 TCP/IP 协议，即网络上所有的节点都执行一个统一协议的开放代码来对等地传输信息。但是这样一个并不复杂的创新对于人类的影响是划时代的。互联



网消灭了过去价值很低、成本很高的信息供应链,使得各国经济体能基于开放、互联、对等的互联网协议联成一体,高效的信息流动推动高效的配置资源,形成了高效的全球统一市场。然而,现有互联网解决了信息的低成本、高效率传输的问题,却没有解决信息的信任问题。信任需要依靠中心化的机构来建立和维持。由此派生出来的问题就是互联网日趋中心化。中心化的问题是容易形成垄断,特别是数据垄断和隐私侵犯问题,同时也会出现单点故障、易受攻击等安全问题。因此,下一代互联网必须突破的是:怎样去中心化地建立全球范围内的互联网信任机制;如何让互联网演进到保证信息、价值安全,传递可靠的价值互联网。

基于信任机制的交易将形成信用,信用的流通可以形成价值。众所周知,中心化的信用,如各国法币,信用价值不同,清算体系也各不兼容,给全球贸易增加了很大成本。目前,以美元为中心的全球信用体系,在机制上存在“特里芬悖论”,即一国法币无法同时兼顾解决本国经济利益和全球经济需要的冲突。而价值互联网正是建立在基于去中心化算法信任的数字货币基础之上,加上区块链是推动分布式大规模协作的最佳技术。因此从社会学来看,将来以区块链的算法信任为基础的数字货币体系,将可以在全球经济体系中扮演重要的价值公平交换角色。区块链作为下一代的互联网应用协议,在数字经济时代数据权益的确立和保护,建立诚信的商业和社会体系方面将发挥重要作用。

和以往的科学技术主要改变生产力不一样,区块链是一种可以改变生产关系的技术。它将信用公证的权利从中心机构转到每个个体手中,建立分布式的透明公共账本,由互联网的各个节点记账和公证,搭建分布式的共识体系来建立信任和监控。区块链的出现预示着下一代互联网,即价值互联网即将诞生,更深层次变革现有社会生产关系,进一步适应和促进生产力的发展。

由于区块链起源于信息技术,我们有必要回顾信息技术发展的历程,从而更好地把握区块链未来的发展方向。

## 1.2 计算模式的演变

在计算机出现以后,计算技术领域经历了3个重要阶段:以大型机等主机为主的集中式计算模式,以微型机为主的桌面计算模式和以网络为主的分布式计算模式<sup>[2]</sup>。计算模式随着计算机科学技术的发展而演进,同时也推动了人类社会由工业革命时代迈向信息革命时代。

### 1.2.1 分布式计算的出现

20世纪60年代,大型计算机的出现开启了信息化革命的前奏。信息革命的深入使得成本昂贵的中央计算模式不能满足经济发展所需要的广泛信息处理需求。随后在20世纪80年代,个人电脑分散计算模式登上历史舞台,极大地推动了信息化技术和应用的普及。客



户/服务器计算模式出现后,个人电脑分散计算模式就从当初的主要计算能力提供者降低为网络客户终端的计算能力提供者。个人电脑的计算能力有限,加上在可靠性、可管理性和安全性等方面都存在很大问题,使其不能适应信息社会发展的需要。用户迫切需要将分散的计算资源连成网络,以较低成本共享计算资源与处理能力,实现大范围协同工作。在这种情况下,分布式计算技术受到关注并得到很大发展,使得计算模式又一次发生变革——从个人电脑的分散计算模式进入了分布式计算模式。

和集中式中心化计算以及个人电脑分散式计算模式不同的是,分布式计算将一个任务拆分成多个任务,放在多个节点上并行执行,这样大大提升了运行效率。但分布式计算也带来一些问题,我们将在下面讨论。

### 1.2.2 分布式计算的挑战

分布式计算系统面临的挑战是:网络环境下,故障情况发生是常态——网络可能会断网,节点可能宕机,特别是公共互联网上还存在着蓄意破坏的节点或者被木马、病毒植入的节点。在这种经常发生故障的环境里面,需要分布式共识算法来解决系统的一致性问题,以保障分布式系统中各节点能够协作。从20世纪80年代开始,计算机科学家们就已经开始了分布式共识算法的研究。科学家们发现,在任何一个有故障节点存在的异步通信网络环境中,理论上找不到一个确定性的共识算法,能在有限时间内让所有非故障节点达成一致。这个看上去非常严苛的理论限制并不能停止科学家们研究共识算法的步伐。相反,科学家们根据实际的应用场景,特别是把异步通信的条件放宽,引入同步、半异步、消息认证机制等网络通信条件,设计出了不同的实用共识算法。

之后科学家也发现,在一个可能发生故障的分布式网络中,不可能同时达到一致性、可用性和分区容错性的目标。

分布式系统的这些局限使得架构师在设计分布式系统的时候,往往需要根据实际情况做权衡取舍。一般来说,传统企业计算强调强一致性,因此所设计的分布式系统需要牺牲可用性以换取强一致性和分区容错性。谷歌GFS系统的设计就采用了这个思路。而另一方面,在互联网那样对可用性和扩展性要求高的分布式环境中,传统计算模式中强一致性至关重要,往往需要弱化成“最终一致性”。例如亚马逊AWS的Dynamo分布式数据库,就牺牲了强一致性来换取可用性和分区容错性。

因此,共识算法也分为“强一致性”共识算法和“最终一致性”共识算法。“强一致性”共识算法又分成可以容忍宕机或宕机恢复故障的Paxos家族共识算法,和可以容忍任意节点故障(也称为拜占庭故障)的BFT共识算法。“最终一致性”共识算法容忍网络状态的暂时性不一致,例如比特币的PoW共识机制。例如在某一时间,比特币网络可以有暂时性的分叉,但随着时间的推移,其协议设计思路是正常的节点会跟随最长的链条,从而解决暂时性不一致的分叉问题。

科学家在分布式共识算法上所取得的成果,加上在密码学和P2P网络上的研究进展,

奠定了后面区块链架构的技术基础。

### 1.2.3 比特币“突围”

#### 1. 比特币的诞生与传播

2008年9月，以雷曼兄弟的倒闭为开端，金融危机在美国爆发并向全世界蔓延。为应对危机，各国政府采取“量化宽松”等措施，救助由于自身过失陷入危机的大型金融机构。这些措施带来了广泛的质疑，并一度引发了“占领华尔街”运动。

2008年10月31日纽约时间下午2点10分，在一个普通的密码学邮件列表中，几百个成员均收到了自称是“中本聪”的人发的名为《比特币：一种点对点的电子现金系统》（Bitcoin: A Peer-to-Peer Electronic Cash System）<sup>[3]</sup>的电子邮件，其中描述了一个新的货币体系。同年11月16日，中本聪放出了比特币代码的先行版本。

2009年1月3日，中本聪在位于芬兰赫尔辛基的一个小型服务器上挖出了比特币的第一个区块——创世区块（Genesis Block），并获得了首个区块奖励——50个比特币。在创世区块中，中本聪写下这样一句话：

“The Times 03/Jan/2009 Chancellor on brink of second bailout for banks”（“2009年1月3日，财政大臣站在第二次救助银行的边缘”）

这句话是当天《泰晤士报》头版的标题。中本聪将它写进创世区块，不但清晰地展示着比特币的诞生时间，还表达着对旧体系的嘲讽。正是比特币的出现使得区块链正式进入了公众视野。

如今，比特币已经成为数字货币领域的翘楚，拥有数千亿美元的市值，但中本聪却于2010年12月选择隐退。中本聪是谁？这是每一个开始了解比特币的人都感兴趣的话题。从《纽约客》到《新闻周刊》，媒体们找到了数个自称是中本聪，或者被认为是中本聪的人。但无一例外，这些发现都因为可信度不足，受到公众的质疑。中本聪是谁，也许我们永远不得而知。

#### 2. 比特币价值发展历程

比特币创始之初，在很长时间内只在技术工程师之间以娱乐为目的进行流通，只能属于一种“玩具货币”。第一笔比特币交易发生于2009年1月12日，中本聪发送了10比特币给密码学专家哈尔·芬尼。2010年5月21日，佛罗里达程序员用1万比特币购买了价值25美元的比萨优惠券，随着这笔交易诞生了比特币第一个公允汇率。自此以后，比特币与实物的交换以这种间接模式慢慢开始。据CoinDesk估算，目前全球大约有60 000个商家接受比特币。表1-1概括了比特币的发展历程。

表 1-1 比特币的发展历程

| 时间       | 事件  |
|----------|---|
| 2008年11月 | 中本聪发表论文《Bitcoin: A Peer-to-Peer Electronic Cash System》，提出比特币概念 |



(续)

| 时间              | 事件   |
|-----------------|--|
| 2009 年 1 月      | 第一个序号为 0 的比特币的区块——创世区块诞生，几天后的 1 月 9 日出现序号为 1 的区块，并与序号为 0 的创世区块相连形成了链 |
| 2009 年 1 月 12 日 | 第一个比特币交易发生，中本聪发送了 10 比特币给密码学专家哈尔·芬尼                                  |
| 2010 年 5 月 21 日 | 第一个比特币实体交易发生，佛罗里达程序员用 1 万比特币买了价值 25 美元的比萨优惠券                         |
| 2010 年 7 月      | 第一个比特币平台成立，新用户暴增，价格暴涨  |
| 2011 年 2 月      | 比特币价格首达 1 美元，英镑、巴西雷亚尔、波兰兹罗提汇兑交易平台成立                                  |
| 2013 年 11 月     | 比特币价格达到历史高峰，盘中高达 1 242 美元  |
| 2014 年          | 中国成为比特币交易增长最迅速的国家，有超过 15 家大型的比特币交易平台处理比特币相关的业务                       |
| 2016 年 1 月      | 比特币逐渐受到质疑，著名比特币挖掘者套现离场   |
| 2017 年          | 被认为是区块链技术的元年   |
| 2017 年 12 月     | 比特币突破 19 800 美元  |
| 2018 年          | 很可能成为区块链技术的爆发年   |

### 3. 比特币的交易

比特币使用整个 P2P 网络中众多节点构成的分布式数据库来确认并记录所有的交易行为。图 1-1 是中本聪在其比特币白皮书中的示例。图中，每笔交易的输入都来自过去未花费的某笔交易的输出（比特币里叫 UTXO），该笔输出有接收人的地址。当所有者 2 要将他的比特币转给所有者 3 时，他首先将他从所有者 1 那里接收到的比特币作为输入，然后将所有者 3 的地址放在交易的输出部分，最后在交易中有自己的私钥签名，并附上自身的公钥。在交易验证的时候，矿工将检查所有者 2 的签名是否和他的公钥匹配。由于每笔交易单都记录了该笔资金的前一个拥有者、当前拥有者以及后一个拥有者，我们就可以依据交易单实现对资金的全程追溯。这也是比特币的典型特征之一。最后，当每一笔交易完成时，系统都会向全网进行广播，告诉所有用户这笔交易的实施。

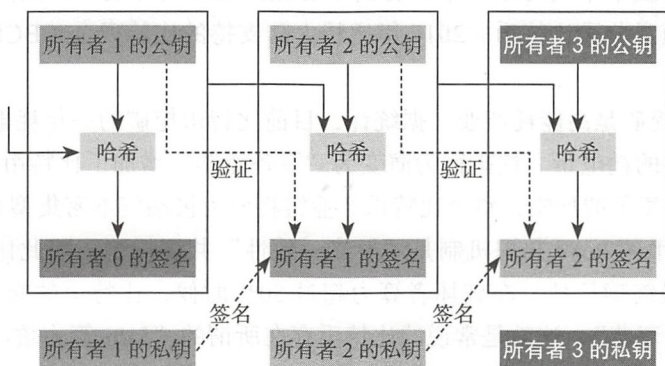


图 1-1 比特币的交易过程示意图

由于每笔交易是相对分散的，为了更好地统计交易，比特币系统创造了“区块”这一



概念。每个区块均包含以下 3 种要素：一是本区块的 ID（哈希值）；二是若干交易单；三是前一个区块的 ID（哈希值）。比特币系统大约每 10 分钟创建一个区块，其中包含了这段时间里全网范围内发生的大部分交易。每个区块中也包含了前一个区块的 ID，这种设计使得每个区块都能找到其前一个区块，如此可一直倒推至起始区块，从而形成一条完整的交易链条。因此，从比特币的诞生之日起，全网就形成了一条唯一的主区块链，其中记录了从创世区块诞生以来的所有交易记录并以每 10 分钟新增一个区块的速度扩展。这条主区块链在每添加一个区块后，都会向全网广播，从而使得每个参与比特币交易的主节点上都有一份拷贝。在现实世界里，每笔非现金交易都由银行系统进行记录，一旦银行计算机网络崩溃，所有数据都会遗失。而在互联网世界里，比特币的所有交易记录都保存在全球上万台计算机中，只要全球有一台装有比特币程序的计算机还能工作，这条主区块链就可以被完整地读取。如此高度分散化的交易信息存储，使得比特币区块链上交易完全遗失的可能性变得微乎其微。

### 1.2.4 比特币的局限

比特币的区块链系统自 2009 年在互联网上开源运行以来，有成功的地方，但也显示出一些难以克服的内在缺陷。

1) 按中本聪的设计，比特币约每 10 分钟出块，每个块的容量定为 1MB，因此理论上每秒只能确认 7 笔交易，这和大部分的商用系统所要求的交易吞吐率有很大的差距。相比之下，Visa 的网络系统每秒最快可处理 10 000 笔交易，支付宝的记录是 2014 双 11 购物节每秒 80 000 笔。因此，当比特币使用者增多时，交易排队等待打包的时间会很长。

2) 比特币当初的设计是“每个 CPU 一票”，强调公平，但随着比特币价格的攀升，挖矿成了暴利行业，矿机厂商开始推出专用芯片（ASIC），通用 CPU 或 GPU 挖矿被淘汰，个人挖矿已经无利可图，矿池越来越集中，到现在全球 70% 的算力集中在前五个最大的矿池中。矿池算力的过度集中使得原来中本聪去中心化的设计名存实亡，因为矿池主的话语权大增，他们可以随意分叉比特币。2017 年比特大陆支持的比特现金（BCH）分叉就是一个典型的例子。

3) 比特币的挖矿是高能耗产业。据统计，目前比特币挖矿的一年耗电量，已经超过爱尔兰整个国家一年的耗电量。挖矿行为演变为“军备竞赛”增加了比特币体系的资源消耗。随着时间的推移和竞争的加剧，整个比特币产业链将成为极端资本密集型行业。

4) 由于比特币的 PoW 共识机制是“最终一致性”共识机制，因此比特币的交易确认基于概率，缺乏最终确认性。在破坏者算力超过 50% 时候，比特币的交易有可能被推翻，比特币有可能被“双花”。这就是常说的比特币存在所谓的“51% 算力攻击”的威胁。在矿池算力高度集中的今天，这个威胁并不遥远。

5) 比特币加大了金融监管的难度。在传统金融监管过程中，几乎所有国家都非常依赖银行系统来查验交易的资金进出。而比特币独立的支付网络则有效地躲开了这种追查途径，

使得国家监管资金动向非常困难。由于其匿名特性，比特币成为犯罪资金的主要载体，在许多非法网站上，甚至成为唯一的支付手段。

## 1.3 区块链的演化路径

### 1.3.1 区块链与互联网意义

一般意义上，我们把第一代互联网叫作“信息互联网”，中国的BAT本质上都是基于信息互联网提供服务；区块链以其分布式账本结构、去中介化信任机制和不可篡改等技术特性，为“价值互联网”时代的到来提供了最核心的技术支持。

区块链是分布式数据存储、点对点传输、共识机制、加密算法、智能合约等计算机技术的新型应用模式，是多种技术有机结合的技术体系。这些技术以新的架构组合在一起，形成一种可信的数据记录、存储、表达和安全的价值转移方式，将成为下一代互联网的核心基础。

### 1.3.2 区块链概念的提出

2014年，比特币已经在无特定机构维护的情况下安全无故障地运行了5年。众多研究学者在总结比特币的成功时，发现了支撑比特币运行的技术体系——区块链，一种集合了共识机制、P2P网络、加密技术、经济学等多种理论和技术的群体智慧机制，为社会各领域展示了一种安全、高效、透明的去中介化社会经济发展模式。2015年，区块链得到了全球各国政府、金融机构的广泛关注。越来越多的专家学者发现区块链在当前社会体制下拥有颠覆性的潜力，并初步将区块链的发展分为3个阶段<sup>[4]</sup>。

第一阶段：区块链1.0，以数字货币为起点的相关应用，主要包括以比特币为代表的虚拟货币，是区块链技术目前最成功的应用。

第二阶段：区块链2.0，由数字资产开启，可以被理解为区块链技术在其他金融领域的运用，如银行间结算、跨境支付、股权登记转让等。

第三阶段：区块链3.0，将区块链应用的领域扩展到金融行业之外，让更广阔的应用场景覆盖人类社会生活的方方面面，涵盖备案管理、知识产权存证、物联网、教育应用和政府管理等诸多方面。在各类社会活动中，区块链可以实现信息的自证明，不再依靠某个第三方机构获得信任或建立信用，从而提高整个系统的运转效率。

### 1.3.3 区块链的社会价值和意义

#### 1. 分布式计算技术进步

传统分布式计算的范式是中心节点将一个任务切分，分配到集群中各个节点并行执行，各节点的执行结果再回到一个中心节点统一协调、组装成最终结果。例如，谷歌在21世纪



初提出的 Map-Reduce 框架就是一个典型的分布式计算范式。在这个基础上，云计算结合虚拟化技术，把分散的物理计算资源池化，按需提供可弹性伸缩的计算资源。这种在分布式系统中加入集中式协同和管控的计算范式能够充分提高资源利用率，提升计算效率和资源使用的灵活性。其缺点是存在着单点故障和在安全攻击下冗余不足、比较脆弱的问题。

和传统分布式计算范式不同，区块链带来了一种全新的分布式计算范式。在这种计算范式中，没有一个强中心来统一协调、管控分布式网络。网络中的节点通过点对点（P2P）通信，消息通过广播而不是经过某个中枢或网关，保证了通信的冗余性和网络的健壮性。更为重要的不同点是，每个任务在每个节点同时运行，运行的结果通过一个共识算法来确认成网络各节点的最终状态。即使网络中存在少数蓄意破坏的节点（拜占庭节点），通过共识算法也能够正常的节点达成一致。虽然这种计算范式的缺点是资源利用率和效率都比传统分布式计算范式要低很多，但这种计算范式的优点在于其健壮性和安全性；更重要的是，它可以避免中心化带来的数据垄断和不公平问题，为数字经济时代的生产关系变革提供了技术基础。

## 2. 解决价值转移的问题

互联网的核心价值是可以让信息高速、低成本地传输，它是一条信息的高速公路。但是它无法传递一类特殊的信息：资产。而区块链恰恰解决了这个问题。区块链为价值的传输提供了技术支持。

在互联网上，我们可以方便快速地生成信息并将其复制到任何一个地方，但我们想将支付的钱直接复制给对方是不行的，必须在付款账户上减去一些钱，在对方收款账户上增加一些钱，才能完成支付过程。这一支付的过程就是一种价值转移。目前的价值转移是由一个中心化的第三方来做背书，比如通过政府、通过银行、通过企业，把所有价值转移的计算都放在一个中心服务器进行处理。区块链则是在没有第三方信用背书的情况下，可以在一个开放式平台上进行线上的安全支付。它跨越多个遍布全球的节点，保存所有交易的历史纪录，网络中所有的参与者都保存着一份完全相同的账本。一旦账本由于新的交易记录而更新，全部副本数据也将在几分钟甚至几秒钟内更新完毕。每一笔交易都有独一无二的时间戳，以防止重复支付。

区块链构建了一种纯粹的点对点的价值转移体系，在不需要各节点互信的情况下，脱离了第三方中心化的背书，并通过区块链的共识机制，防止价值的重复转移（所谓的“双花”）。

## 3. 解决中心化的信用风险和信用成本的问题

现代人类社会最显著的特点是大规模的陌生人群体之间的协作。无论是我们日常的网上购物还是募捐众筹等慈善行为，我们依据什么去相信一个陌生人？这些活动过程中，我们潜移默化地把信任托付给了国家机构或大型企业，这是现代社会的信任方式。而区块链用算法证明机制来建立这份信任。借助它，整个系统中的所有节点都能够在信任的环境下



自动安全地交换数据、自动撮合、强制执行交易，从而降低了建立信任的成本和防范信用风险的成本。

如果上升到哲学层面，区块链所信奉的是“相信人，不如相信技术”。区块链技术可以将公信力抽象出来作为一个独立的存在，而不是由某一组织或机构掌控。在信息不对称、不确定的环境下，区块链通过去中介化技术，能够在陌生节点组成的网络环境中完成数学（算法）背书、全球互信，建立一个全国乃至全球性的信用共识体系，同时将经济活动发生的相关机构联合起来建立联盟链或公有链，使得彼此信息有权限地共享，交易信息不会被篡改，让各利益主体彼此信任，从而在人类历史上实现多中心化分散式的大规模信用机制，在消除中心机构“超级信用”的同时，保障了信用机制的安全、高效运行，同时奠定了人类大规模协作的技术基础。

因此可以说，区块链是创造信任的机器。信任是一切契约达成的基础，契约的建立和执行则是一切商业和金融活动的基础。在信息化社会和数字经济时代，数字化的契约可以形成资产，而资产具有价值的属性。基于信任的契约的履行可以形成信用。区块链作为不可篡改、可追溯的共享账本提供了完整的信用数据。所以，区块链解决了信任建立、信用共识的问题，同时也提供了安全的数字化资产价值转移的载体，成为数字经济时代最重要的技术基础设施。

#### 4. 社会经济模式升级

##### （1）一个全新的世界账本

据考古学家考证，人类最早的文字创造来源于记账的需求。从遥远的石器时代的石刻与结绳记事，到形成于古希腊、古罗马的商业经济雏形的单式记账法；从15世纪欧洲商人广泛使用的复式记账法，再到今天让电脑、机器人参与记账，这一系列大变革所呈现出的工具、技术、方法更替，始终没有解决现代会计依然存在的几个问题：

1) 现代公司治理中，所有者和经营者分离，对账本的需求越来越复杂，而无论是哪种记账方法都是基于事后的数据记录；

2) 当前可靠性是依靠管理人员或记账人员承诺他们的账本没有问题，诚信之外还需要第三方信用中介进行公信力的背书；

3) 人为性出错是会计问题的主要原因；

4) 现代公司复杂度的增加，让隐藏错误的行为变得更加容易了。

区块链被称为“一个不停地进行审计和验证的公共账本”<sup>[5]</sup>。它将诚信融入每一笔交易当中，在涉及多实体的商业活动中，也许不再像复式记账那样需要事务所的中介审计和信用背书，而能自动提供对账、清结算功能，同时可以让股东、审计及监管者即时访问账本，提供透明性和客观公允性。这是区块链可能给商业会计制度带来的重要价值和变革的动力。

##### （2）区块链将缔造一个崭新的共享经济

区块链天生就具备去中心化的特性，这一点与共享经济的宗旨有着高度的吻合。区块

链作为一个去中心化的一致性共享数据账本,在其架构下,整个系统的运作都是公开透明的,它将让共享经济变得更加公平、透明。同时区块链的共识算法能在社群经济中形成社群协作的信任基础。区块链通过借助智能合约技术,能够自动执行满足某项条件下的操作,自动实现经济激励和利益分配,大幅降低契约建立和执行的成本。

总体看来,在政府、行业巨头、创业者和资本的共同推动下,区块链将会以更快的速度向前发展,它将与人工智能、大数据等新技术一起塑造一个全新的经济体系。

## 1.4 小结

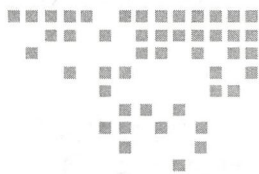
互联网发展到下半场已经出现诸多问题限制其快速发展,给现有产业带来隐患。区块链多中心、分布式的存储与决策架构,能够有效解决当前互联网不能处理的价值转移问题,同互联网一样,未来区块链将会深入应用到社会经济和产业的各个角落,对互联网下半场的社会经济和产业带来不可估量的价值与意义。

### 1. 习题

- [1] 列出传统分布式计算范式与区块链分布式计算范式的异同点。
- [2] 简要概括比特币的发展历史及未来影响。
- [3] 试分析区块链引起的技术变革和深远影响。

### 2. 参考资料

- [1] 熊江. 分布式计算模式的演变与发展 [J]. 重庆三峡学院学报, 2007, 23(3):31-33.
- [2] 陈宏, 张超, 张玉山. 浅议计算机计算模式的演变及发展趋势 [J]. 青岛远洋船员职业学院学报, 2013, 34(2):18-22.
- [3] NAKAMOTO S. Bitcoin: a peer-to-peer electronic cash system [EB/OL]. <https://bitcoin.org/bitcoin.pdf>.
- [4] SWAN M. Blockchain: blueprint for a new economy[M]. USA: O'Reilly Media Inc, 2015.
- [5] 唐塔普斯科特, 亚力克斯·塔普斯科特. 凯尔·孙铭, 周沁园, 译. 区块链革命: 比特币底层技术如何改变货币、商业和世界 [M]. 北京: 中信出版社, 2016.



## 区块链基本概念

作者：邢春晓 张桂刚

区块链是随着比特币等数字加密货币的发展而逐渐被人们所熟知的。区块链是一种分布式数据存储机制，是一串使用密码学方法链接的数据块，每一个数据块中记录了一批当时网络交易的信息，包含用于验证其信息的有效性（签名防伪）和生成区块的协同信息（例如工作量证明的难度目标）。比特币区块链信息在网络上公开的，可以在比特币浏览器中查询过往的交易数据。比特币钱包提供用户的当前余额信息，其功能依赖于与网络节点对比特币交易的共识确认。比特币上的区块链每增加一个区块，就可以看作是一次共识确认。在比特币上，通常一次交易要获得 6 个区块确认才能很大概率上保证生效。区块链本质上是一个分布式账本数据库，是比特币的底层技术，和比特币是相伴相生的关系。为了更好地帮助大家理解区块链相关技术，并形成系统性的区块链的技术架构，本章首先以比特币为例介绍区块链的第一个用例（Use Case），帮助大家理解区块链，理解为什么需要研究区块链技术；研究区块链技术的社会原动力在哪里。然后再逐一详细介绍区块链的相关技术以及未来应进行的标准研究。

### 2.1 区块链技术研究缘由

区块链之所以获得全球的关注，原因在于区块链技术可以带来社会迫切需要的去中介化信任机制，改变社会的各种应用的计算范式，并为社会带来巨大的效益。根据 Gartner 的研究预测，Blockchain 将在未来 5 ~ 10 年迎来爆发式发展，并得到实际应用。图 2-1 为 2016 年 7 月 Gartner 发布的最新的未来技术的预测图，其中区块链技术（Blockchain）就包含在其中。为了让大家清楚为什么要研究区块链技术，首先让我们从分析区块链技术的



第一个用例——比特币入手。对比特币的初步了解，将有助于我们了解区块链技术的重要性。

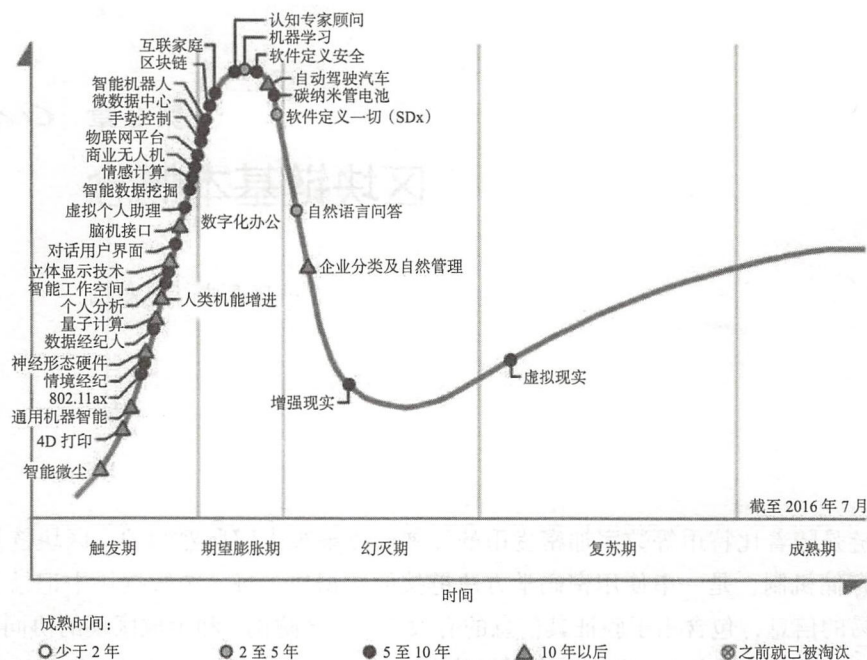


图 2-1 区块链技术的未来

### 2.1.1 区块链用例描述：比特币

互联网上的电子商务，几乎都需要借助金融机构作为可信赖的第三方来处理电子支付信息。而比特币的出现，可以摆脱第三方的限制，完全依靠算法来确保交易的可靠性。它完全不同于传统的物物交换、基于金银的贵金属交换、基于政府信用背书的纸币作为流通货币的交换、基于政府信用背书的电子货币交换，而是完全依赖于算法本身的可靠性，将引领一种新的交易范式。

假设有 A、B、C 三个人正在使用比特币，并希望通过比特币进行交易。他们需要解决三个问题：第一是来源问题，即如何获得比特币；第二是解决怎么交易比特币问题；第三是怎么保证交易安全可靠，怎么确认获得的比特币是真的，也就是解决防伪防欺诈的问题。下面我们来看中本聪的设计是如何解决这些问题的。

#### 1. 怎么获取比特币（来源问题）

A、B 以及 C 获取比特币的来源有两种途径，分别为：按照中本聪的想法，第一种方法可以比较形象地表示为挖金矿，自己当矿工，通过算力挖比特币；第二种办法是通过交易或者别人捐赠获得比特币。

### (1) 当矿工获取比特币

中本聪挖到了世界上第一个比特币区块，也被称之为“创世区块”，从而得到了 50 个比特币的挖矿奖励。每个区块（Block，账本）的创建者都将得到相应的比特币。就像现实世界中的矿产资源有限、矿的优劣程度也不同一样，中本聪对应制定了如下的挖矿规则：前 21 万个区块的初始值为 50 个比特币，即前面 21 万个区块的创建者每创建一个新的区块均可以得到 50 个比特币。这可以通俗地理解为前 21 万个区块为最优质矿。产生到第 21 万个区块之后，每个区块的创建者只能得到 25 个比特币。依此类推，第 42 万个区块创建后，区块里初始只有 12.5 个比特币。也就是说，随着优质矿被挖走，矿产质量越来越差，价值也越来越低。矿产终究有挖完的一天，所以最后比特币的钱总数会停留在 2100 万个。矿挖完了，但是交易永远都不会停止，因此，区块会不断地创建下去，只不过后面那些挖矿的人即使创建一个区块，也不会再得到比特币的挖矿奖励，但可以获取区块中所有交易的交易费用。

根据中本聪的设计，世界上约每 10 分钟就会产生一个新的区块，不管这个区块是否有交易，都会约每 10 分钟产生一个。如果有很多矿工想去挖矿（挖矿算力增加），则算法会动态地调高出块的难度来保证出块节奏稳定在大约 10 分钟出一块。所以，通过挖矿得到比特币不是一件十分容易的事情。如果在这十分钟内有 1 万人在挖矿，那也只有一个幸运者能够获得该矿，并得到系统所给予的挖矿奖励。当然，世界上发生在这 10 分钟内的大部分交易，会记在这个区块里面。到一定时间后（2140 年），所有矿产都挖完了，后面每一次挖矿（创建 Block）将不会有任何收益，这将会影响矿工的积极性。因此，中本聪也规定，对于一些非常复杂的交易会收取交易费，这个交易费用将会用于奖励挖矿者。这在一定程度上也提升了矿工的积极性。

回到前面的描述，A、B、C 三人要想获得比特币，首先要成为矿工去挖矿，获得比特币系统给予他们的奖励（每挖出一个最优质矿将获得 50 个比特币，挖到一个次优质矿将得到 25 比特币，依此类推）。

### (2) 通过交易或者捐赠获得比特币

当然，要想获得比特币，去当矿工挖矿不是唯一的途径，也可以通过交易或者捐赠获得。假如 B 是一个猎人，A 可以花 0.5 个比特币购买 B 的一头野猪，一旦这个交易完成，B 就从 A 那里得来 0.5 个比特币。又如 A 有比较多的比特币，为了做慈善，可以为社会进行捐赠，比如给壹基金捐赠 100 个比特币等，这时候壹基金就从 A 那里获得了 100 个比特币。

## 2. 怎么交易比特币

通过比特币进行交易，其实和传统的商品交易没有本质上的区别。唯一不同的是，通过人民币或者美元进行交易既可以通过电子形式（比如通过转账）完成，也可以直接使用钞票（人民币或者美元）去购买所需要的商品。而通过比特币进行交易，只能使用电子形式进行，也就是只能通过账本中进行登记和计算的方式来完成交易。假如 B 有 100 个比特币，A 有 2 000 个比特币，A 花了 0.5 个比特币从 B 那里购买了一头野猪，在账本中记录下了这



笔交易。B 的钱包就会根据账本信息在 B 的账户余额增加 0.5 个比特币，变成 100.5 个比特币；而 A 的钱包根据账本信息在 A 的账户余额减少 0.5 个比特币，变成 1 999.5 个比特币。从这里可以看出，比特币所有的交易都体现在账本里面。

### 3. 怎么确保交易安全可靠

现实中的面对面交易，一手交钱一手交货，只要钱不是假币，整个交易将会很顺利，并且双方都会很满意，可以马上完成交易。现实中的电子交易有中央银行的 CA 中心（国家机器）作为信用背书者，CA 会确认电子交易双方的真实性（CA 会证明 A、B、C 是实实在在存在的真实交易者），并且防止他们抵赖（比如：A 下了购买 B 的野猪的订单，但是对购买行为不承认；或者 B 收到了 A 购买野猪的钱，但是不承认已经收到，拒绝通过物流运输野猪给 A；等）。比特币交易系统不一样，它没有任何组织（欧盟发行的欧元）或者国家（美国发行的美元、中国发行的人民币）来背书，交易安全完全需要依靠技术手段本身来解决，比如依靠各种加密手段、验证手段、算法等。具体可以归纳为如下几点。

#### （1）如何确保比特币是真实的

比特币与现实中的美元或者人民币不一样。现实货币的真假可以通过银行或借助验钞机等方式来判别真假，而比特币是一种虚拟货币，无法通过识别水印或者特殊标记、制作材料等来进行判断，其真实性的判断依据是它的来源是否真实。人们获取比特币只有挖矿（创建账本）和通过交易或者接受捐赠获得两种方式。所以判断比特币是否真实，只需判断其是否有真实的来龙去脉，是否被“双花”过。比如在上面的例子中，当 B 获得 A 的 0.5 个比特币的时候，他怎么确认是真的呢？首先比特币区块链采用密码学的非对称加密公钥系统的签名来保证 A 的交易来自于 A，同时 A 的比特币有来龙去脉。就像前节所述的，A 的比特币要么是自己挖矿挖来的，要么是别人转给的，总之 A 的比特币不能凭空产生，其来源可以被验证。

#### （2）如何防止重复支付（“双花”）

在传统的面对面交易中，如果 D 只有 1000 元人民币，那么他用这 1000 元人民币购买了一个手机后，就不可能再用这 1000 元人民币去购买自行车，因为这 1000 元人民币在购买手机的时候就已经支付给了手机销售商。同样，在传统的电子商务交易中，如果 D 的电子银行账户中只有 1000 元人民币余额，那么当 D 在某个时间同时下单购买价值 1000 元人民币的手机和价值 1000 元人民币的自行车的时候，交易中心会确认哪一笔买卖在前、哪一笔买卖在后，如果购买手机的时间在前，交易中心会通知数据库转账 1000 元人民币给手机销售商的银行账户，通过数据库的锁定机制，钱就不可能再支付到自行车销售商的账号里，也就是说也不可能出现重复支付的问题。但是在比特币的交易系统里面，由于没有中心化信任机构，因此，交易需要广播给参与账本的所有人，需要所有人进行确认，这就需要一定的时间来完成。可能“D 购买价值 1000 元手机”的交易还没有确认完，“D 购买价值 1000 元的自行车”的交易就已经开始确认了。于是就有可能出现两笔交易同时有效的现象（即“双花”）。但实际上 D 只有 1000 元，却购买了价值 2000 元的东西。这是比特币交



易体系里面一个不能容忍的现象，因此需要借助“共识机制”来解决该类问题，从而确保交易的先后顺序、确认哪一笔交易是有效的，等等。比特币采用工作量证明机制来保证不能出现“双花”。B 只要看到 A 转给他的 0.5 个比特币有 6 个以上的区块进行了确认，就可以有很大把握确认这 0.5 个比特币不可能被 A “双花”。

### 2.1.2 区块链需要研究哪些关键技术

区块链将成为未来计算范式的一个基石。为了促进区块链技术的发展，到底需要研究哪些关键技术呢？通过前面的分析，我们对区块链有了一个大体的了解，区块链关键技术的研究应该包含如下几个方面。

#### （1）区块链共识机制

为了确保交易的有效性，区块链主要是通过交易参与者对账本中交易的统一确认来完成的。因此需要研究不同的应用场景所需要的各种共识机制，比如，也有解决强一致性问题共识机制，也有解决“最终一致性”问题的共识机制，如工作量证明 PoW、权益证明 PoS、股份授权证明 DPoS 等。目前，各种共识机制都存在缺点，例如比特币的 PoW 机制存在高耗能、无用功和性能低下的问题，而 PoS 则存在安全性低的问题。随着应用的不断丰富以及复杂化，如何寻找满足各种不同应用场景需求的，更高效、更安全的共识机制是业界需要持续关注的问题。

#### （2）区块链的安全与隐私保护技术

与传统的计算范式完全不同，区块链不再借助中心化机构来确保信用可靠，其安全性完全通过技术手段来解决。因此，研究区块链的安全性保障显得至关重要，包括数字加密技术、数字签名、身份认证、授权鉴权、隐私保护、审计追溯、防止网络安全攻击等技术。

#### （3）区块链的存储技术

区块链是一个由众多的区块（Block）组成的相互关联的账本链。随着区块链数据的增加，区块链的数据如何在云计算环境下进行高效存取，如何确保存储与计算效率，存储在文件系统还是数据库，如何存储在分布式数据库或者分布式文件系统，如何提升区块链的查询效率，什么数据存放在链上、什么数据存放在链下的数据库，如何压缩区块链的数据以节省空间等问题，都是区块链存储必须关心的问题。

#### （4）区块链的通信技术研究

区块链主要是运行在分布式环境下。如何高效地同步数据？如何提升通信的可靠性？如何防止 DDoS 攻击？如何提升广播效率？如何对通信进行有效验证？这些问题都是区块链在通信过程中需要研究的。

#### （5）区块链的智能合约

最主要的是要研究提供智能合约的计算引擎，通常是虚拟机的技术。这其中包括虚拟机的效率、安全机制、图灵完备性、高级语言编译器、智能合约形式化证明、智能合约的升级、治理机制等。随着区块链 3.0 应用的需要，还需要研究能够满足人类社会管理所需

的、更为复杂的、具有语意功能的复杂应用算法库以及算法组合机制等。

### （6）区块链的应用体系

比特币是区块链技术的一个成功应用，但它仅仅是一种虚拟货币支付体系的应用。其实区块链技术还可以应用到社会的各个方面。区块链技术发展可以分成三个阶段：1.0 阶段、2.0 阶段和 3.0 阶段。这些在第 1 章有过讲解，这里不再赘述。

### （7）区块链技术标准

一项技术要想实现产业化，形成生态产业链，标准化是其必经之路。区块链技术也不例外。目前区块链技术在国内外尚未形成统一的标准。区块链技术及其具体的应用过程会涉及各种技术标准，逐步开发与区块链相关的技术标准，将有助于实现区块链技术的成熟，从而令它成为一个可以工业化、可以应用到实体经济的、真正意义上的实用技术。

## 2.2 区块链模型

比特币是区块链的一个最为成功的应用。基于区块链技术模型和该模型所包含的各种关键技术，除了可用于比特币之外，还可以用于互联网其他新应用的方方面面，从而将区块链技术推广应用于社会，为社会服务。我们通过调查研究，将区块链模型总结为图 2-2 所示。

从图 2-2 中可以看出，区块链技术模型包括 9 大部分，其中包含 7 层基础技术层以及 2 个贯穿整个 7 层的共用技术。9 大部分分别为：数据存储层、网络通信层、数据安全与隐私保护层、共识层、智能合约引擎层、应用组件层、区块链应用层、区块链与现代技术融合以及区块链技术标准。

### （1）数据存储层

形象地说，区块链就像一个个分布式账本，账本的账册之间通过链条链接在一起，构成一连串的账本链（即区块链）。账本里的数据需要以分布式方式存储在不同的节点，其存储形式需要借助于分布式文件系统或者分布式数据库技术来完成。因此，数据存储层主要包括数据区块的逻辑组织方式以及如何有效地实现对分布式账本的存储。

### （2）网络通信层

区块链的区块数据和交易数据需要通过 P2P 网络在不同的节点之间进行同步、验证，这就需要网络通信层实现数据同步、校验等消息传播机制和验证机制等。

### （3）数据安全与隐私保护层

区块链的安全，需要通过数据安全与隐私保护层来进行保护。该层主要技术包括：密码学加密技术、哈希算法、数字签名技术、身份认证技术、授权鉴权技术、零知识证明等隐私保护技术、防范网络攻击技术、审计追溯技术以及抗量子安全算法等安全保障技术。与传统的中心化 PKI 安全体系不一样，区块链上的安全技术强调采用“去中心化”的区块链安全技术体系和隐私保护体系。



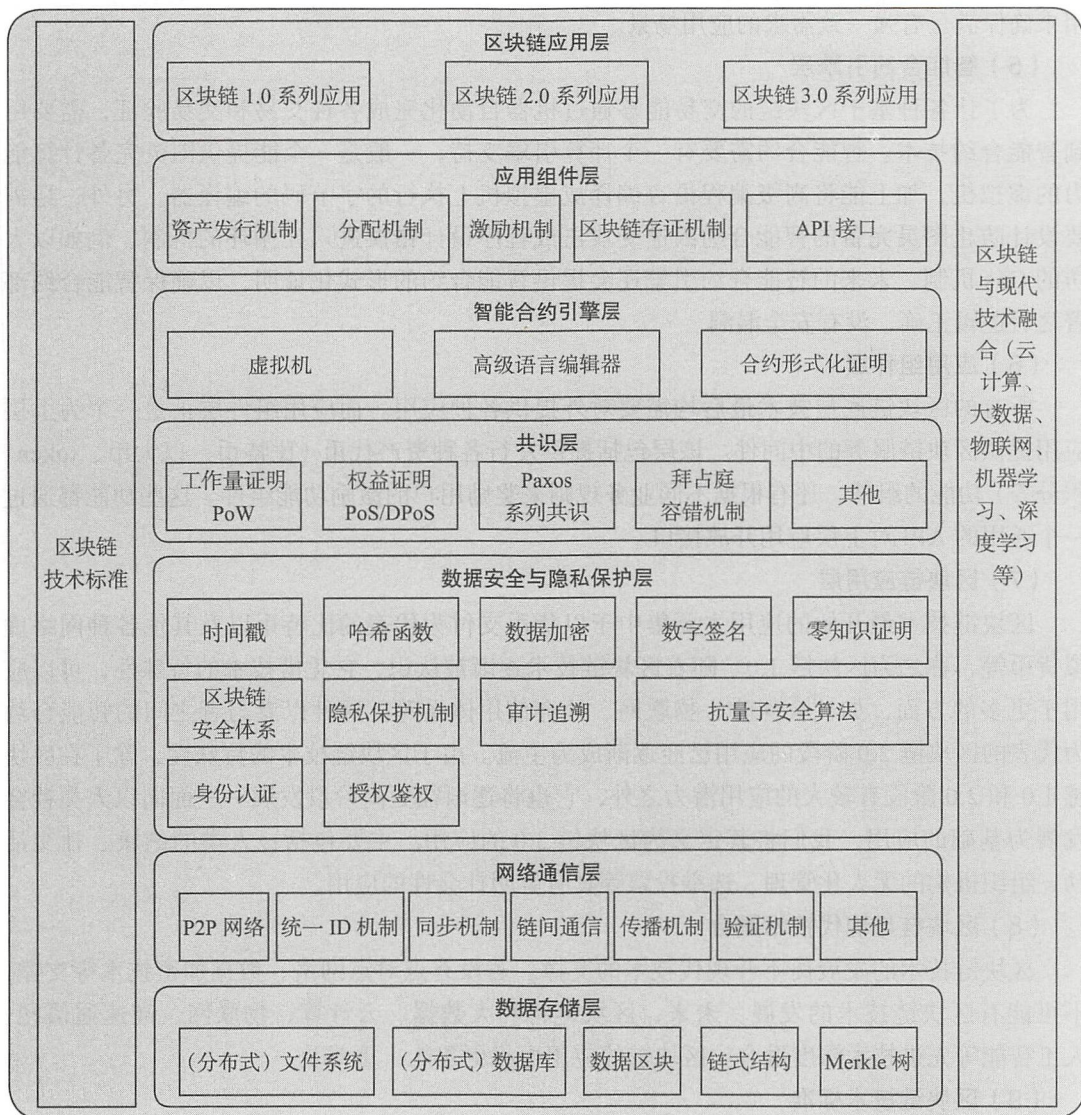


图 2-2 区块链技术模型

#### (4) 共识层

基于区块链的各种应用，其实质是 DApp（Decentralization Application，去中心化应用）。去中心化应用在网络中的各个节点同时运行，其结果需要通过共识机制来形成共识，使得 DApp 应用状态在区块链网络中得到确认。比特币区块链和以太坊的共识层是使用工作量证明（PoW）共识机制，随着应用的不断丰富，通过工作量证明来达成共识已经不能满足应用的需求，特别是有高并发需求的应用场景。因此，在不同的应用场景需要构建能满足应用需求的共识机制，比如后来发展的权益证明（PoS）、股份授权证明（DPoS）等，还有一些传统的强一致共识算法，像 Paxos 家族共识算法和拜占庭容错算法（BFT）等，可以



用来确保满足有强一致需求的应用场景。

### （5）智能合约引擎层

为了让各种基于区块链的交易能够通过机器自动化完成各种交易和交易验证，需要用到智能合约技术。智能合约需要有一个计算引擎支持，一般是一个能提供图灵完备计算能力的虚拟机，加上能将高级编程语言编译成虚拟机上执行的字节码的编译器。另外，还需要设计防止图灵完备的智能合约因遭受攻击或程序设计错误进入死循环的机制，例如以太坊的 Gas 机制。未来的智能合约引擎还会提供智能合约的形式化证明，以确保智能合约部署之后逻辑正确，没有安全漏洞。

### （6）应用组件层

所有的区块链底层技术最后均需要对外提供各种应用。而应用组件层正是一个为上层应用提供区块链服务的中间件。该层包括提供发行各种资产代币（比特币、以太币、token、积分等）功能的组件，还有根据不同业务规则来奖励用户的激励功能组件。这些功能都通过一个通用的 API 对上层应用开放接口。

### （7）区块链应用层

区块链技术最开始的应用主要集中于以货币支付为代表的比特币以及其他各种网络虚拟货币等，称之为区块链 1.0。随着区块链技术逐渐被认识，它凭借技术的特殊性，可以应用于更多的方面，如社会保险、物联网、社会信用体系等。这种以参与者之间的智能合约为代表的区块链 2.0 阶段的应用已经逐渐成为主流。由于区块链技术的特殊性，除了在区块链 1.0 和 2.0 阶段有较大的应用潜力之外，它也将逐渐往更高阶段发展——面向以人类社会发展为基础的应用，我们将其定义为区块链 3.0 的应用。主要包括：人类的健康、社交活动、组织机构的无人化管理、选举投票等具有鲜明社会性的应用。

### （8）区块链与现代技术融合

区块链技术的发展离不开现代技术的支撑。若没有点对点网络、数据加密技术等支撑，不可能有区块链技术的发展。未来，区块链将和大数据、云计算、物联网、高速通信网、人工智能等先进技术深度融合。区块链的应用也将更智能，更普及。

### （9）区块链技术标准

目前区块链技术在国内外尚未形成通用的技术标准。区块链技术涉及众多的核心技术，也涉及众多的数据和数据、应用和应用等的交互操作。标准化工作是一项技术能否通用、能否大范围应用的必经之路。因此，为了加快区块链技术的发展，制定各种区块链技术标准已经刻不容缓。有了统一的标准之后，大家才会对区块链有统一的认知，各种基于区块链的应用才会采用不同厂家互相认可的技术手段，使得不同商家的技术之间可以互相兼容，各种链上的信息和资产可以互联互通，从而促进区块链技术的健康发展。

## 2.2.1 数据区块

数据区块记录了该区块创建期间所记载的所有交易信息，例如比特币的每一个数据区

块将记载某一个 10 分钟时间段内大部分的交易信息。其他应用的数据区块可以各自定义区块的时间范围等。比特币里的数据区块是指比特币交易的账本。其他区块链应用的数据区块是指各自应用所处理的各种交易的详细记录信息。一般来说，所有的数据区块都必须保存在每一个参与者的电脑、服务器甚至云环境中，各个节点都是完全对等的，均保存了数据区块的完整信息，一个数据节点遭到破坏，并不会影响整个系统数据区块的完整性和安全性。当然，根据应用场景的特点，不一定每个节点都保存所有区块的完整信息，例如手机上或者是浏览器上的区块链应用就不会保存所有的区块信息。

数据区块由区块头和区块体两部分组成。区块头保存着各种用于链接上一个区块的信息、各种用来验证的信息以及时间戳等信息，主要包括：块编号、前一个区块头的哈希值、一个用于证明工作量难度的随机数和难度目标、时间戳、用于验证区块体交易的一个总的哈希 Merkle 树根。区块体主要包含该区块（账本）中的所有交易信息以及所有交易信息的 Merkle 树（树根除外，树根存储在区块头内）。图 2-3 简要展示了比特币的数据区块的基本结构。

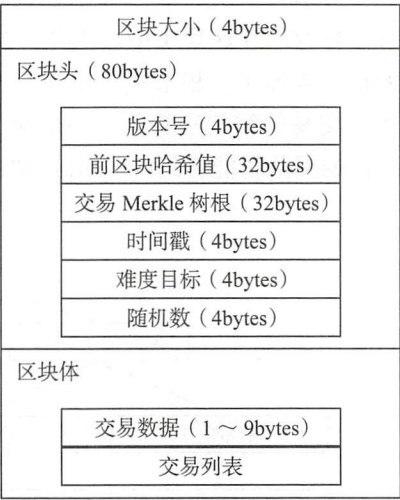


图 2-3 比特币区块基本结构

2.2.2 链式结构

区块链应用的所有区块之间按照时间先后顺序链接成一个完整的链条。该单向链条可以逐渐增加区块，当一个新的区块创建后，就补充在最后一个区块后面。同时该单向链条也可以回溯发生的所有交易信息，从而确保安全性和可验证性。图 2-4 展示了一个比特币的链式结构。

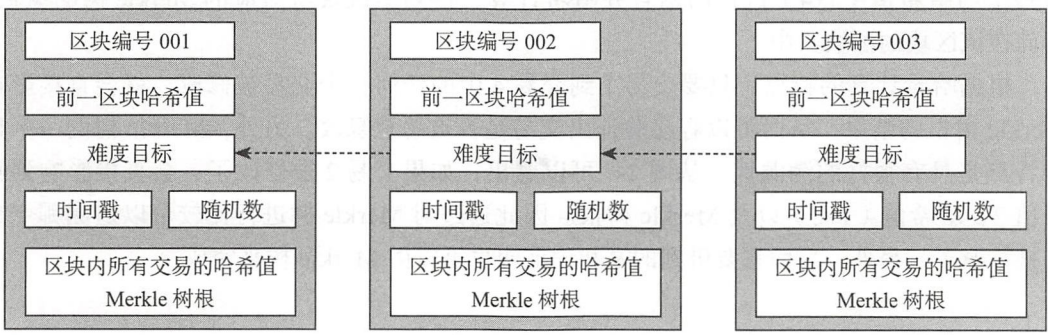


图 2-4 一个简单的链式结构

所有的区块链连接成一长串，应用的所有交易信息都保存在区块内，并且通过链条串



联在一起，每一笔交易都可以向前溯源，找到其所有的历史记录。由于该长串链条由每个节点所认可，有人想要篡改链条几乎是不可能的事情，他必须要将所需篡改的区块以及前后的所有区块都进行修改才可以。否则通过链条的溯源机制，问题很快就会被发现。而篡改链条上的所有区块是不可能的事情，因此区块链的链式结构能够有效地防止数据被篡改。

### 2.2.3 Merkle 树

Merkle 树是一种典型的二叉树或多叉树，它包含根节点、中间节点以及叶子节点。根节点将存储在区块头中。因此，要判断区块体交易数据是否有修改，只需要验证 Merkle 树的根节点即可。根节点、中间节点和叶子节点是一组哈希值。叶子节点存储了该区块内的所有交易的初始哈希值，一个交易对应一个叶子节点。图 2-5 展示了一个简单的、基于区块链的保险智能合约的 Merkle 树的数据结构。

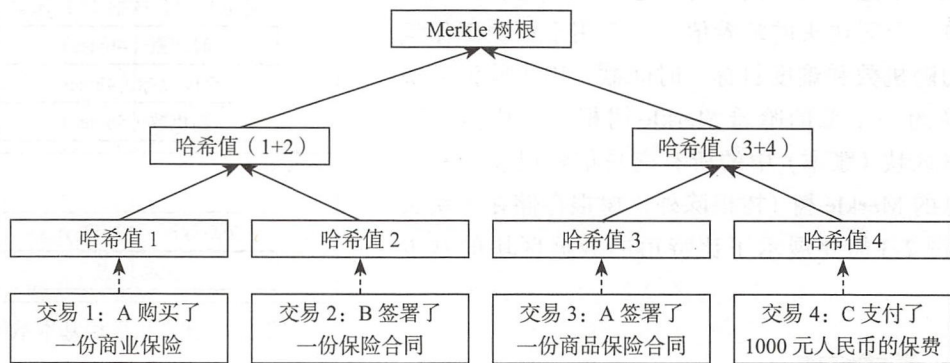


图 2-5 Merkle 树示例

在本区块内有 4 笔交易，每笔交易对应 Merkle 树的一个叶子节点。使用哈希函数对每笔交易进行计算，分别得到哈希值 1、哈希值 2、哈希值 3 以及哈希值 4。然后通过对每个哈希值进行两两合并哈希，分别得到哈希值 (1+2) 以及哈希值 (3+4)。最后哈希值 (1+2) 与哈希值 (3+4) 进行两两合并哈希计算，得到本区块所对应的 Merkle 树的树根，存储在该区块的区块头中。

根据哈希计算的特点，只要交易 1 到交易 4 中的任何一个交易被修改，就会直接影响 Merkle 树根的结果，从而可以容易验证出交易是否被恶意篡改。另外，Merkle 树的一个显著特点是具有很好的溯源性。从图 2-7 可以看出，如果交易 2 被修改了，会直接影响到哈希值 2、哈希值 (1+2) 以及 Merkle 树根。因此通过对 Merkle 树进行比较可以追溯哪个交易被篡改了。另外，后面将要讲到的零知识证明也可以由 Merkle 树来完成。

## 2.3 网络通信层关键技术

和传统应用运行在有中心化的服务器节点或集群环境不一样，区块链应用运行在去中



心化的分布式网络，一般采用 P2P 的组网方式。P2P 的组网方式决定了区块链具有与传统网络不同的网络协议、消息传播方式以及数据验证机制。各种应用正是通过网络层的消息传播机制和数据验证机制，来确保区块链应用中的每个参与者（节点）都能参与区块链中的交易校验和创建数据区块。

### 1. P2P 网络

P2P 是 peer-to-peer 的缩写，又称“对等网络”，可以简单地定义成通过直接交换来共享计算机资源和服务，对等计算模型应用层形成的网络通常称为对等网络。在 P2P 网络环境中，成千上万台彼此连接的计算机都处于对等的地位，整个网络一般不依赖专用的集中服务器。网络中的每一台计算机既能充当网络服务的请求者，又能对其他计算机的请求做出响应、提供资源和服务。通常这些资源和服务包括信息的共享和交换、计算资源（如 CPU 的共享）、存储共享（如缓存和磁盘空间的使用）等。图 2-6 展示了一个区块链环境下的 P2P 网络架构。当然，不同的区块链平台的网络协议可能各不相同，例如比特币、以太坊和超级账本都有自己特殊的网络协议，公有链、联盟链、私有链也可以有各自的网络协议等。

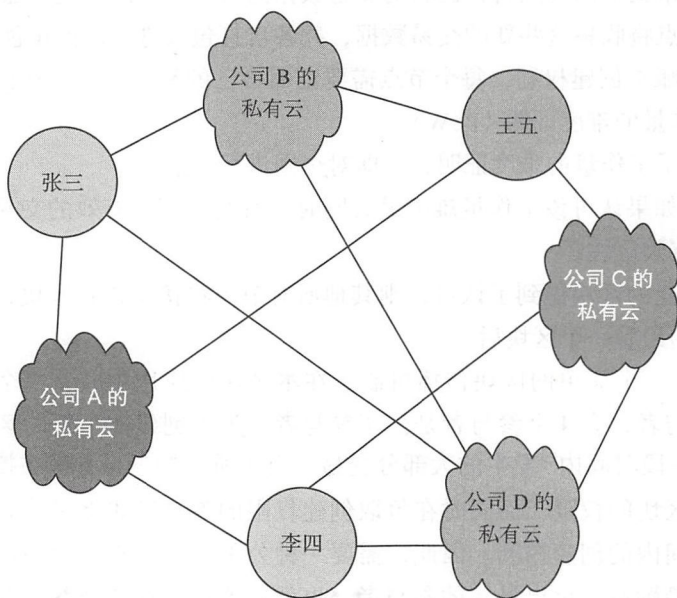


图 2-6 一个区块链应用的 P2P 网络示意图

从图 2-8 中可以看出，任何参与者（可以是单独的人或者公司）都可以是 P2P 网络中的节点。

理论上，所有的区块需要在所有的参与者中都存储一份，也就是说，图 2-8 所示的 P2P 网络中的每个节点都保存区块链中所有区块。但是，通常即使在 P2P 网络中，节点还分全节点和轻量级节点。全节点具有足够的资源，例如公司 D 的私有云，可以存储区块链的所



有区块，而一些本地资源少的轻量级节点可能只存储一些关键的区块（例如张三的手机）。那些全节点可以自行完成各种交易的完整性验证，而轻量级节点需要从其他全节点处调用一些所需区块来帮助自己完成数据的验证工作等。

## 2. 链接方式

当矿工生成一个有效区块，且被其他矿工确认有效后，就可以链接到当前区块链的末尾，形成新的区块链主链。但区块链并不完全以线性方式延长，有时会出现分叉。这是因为区块链系统中，各计算节点是以分布式并行计算来争取记账权的，所以可能会出现短时间内有两个计算节点同时生成有效区块的现象。此时，区块链系统选择将两个有效区块都链接到当前主链的末尾，这就形成了分叉。针对这种情况，区块链协议规定：当主链分叉时，计算节点总是选择链接到当前工作量证明最大的分支上，形成更长的新主链。

## 3. 传播机制

一旦一个新的区块创建后，生成该区块的节点需要将该消息广播给其他所有节点。不同的应用可以设计不同的传播机制。比特币的传播机制可以简要描述如下。

- 1) 发起比特币交易的节点将自己新的交易数据向所有的全网其他节点进行广播。
- 2) 所有的节点将收集这些新的交易数据，并各自打包放进自己的预创建的区块中。
- 3) 为了争夺账本创建权利，每个节点需要证明自己的努力工作，在比特币世界里使用的是找到一个工作量的难度证明（PoW）。
- 4) 一旦找到了工作量的难度证明，立即对全网进行广播。
- 5) 其他节点如果认可该工作量难度最大同时所有交易都是有效的交易，那么认可该节点创建的区块为有效区块。
- 6) 一旦新创建的区块得到了认可，则其他所有节点将接受该新区块，并将该区块加入到自己区块链条的最后一个区块后。

图 2-7 展示了一个简单的区块传播机制。在本区块的应用中共有 7 个参与者，有 3 个参与者是个体参与者，有 4 个参与者是公司参与者。为了创建一个新的区块链，首先所有的参与者要搜集一段时间内<sup>①</sup>发生的大部分交易。由于所有的参与者都在搜集新的交易，希望自己夺得创建区块的权限，而同时在争取创建权限的参与者非常多，只有一个幸运者能够得到某一段时间内的创建权利。因此，需要一种公平的方法来帮助完成此项工作。创建权利将交给工作最勤奋、付出最多的参与者。也就是说在上述 7 个参与者中选择一个最努力的人。比特币采用了工作量证明的方法，给定一个随机数，让所有的参与者去找这个随机数（找这个随机数的难度很大），只有找到该随机数的人才能获得该区块的创建权。当然，机器计算能力越强的参与者获得创建权的可能性越大。我们假设公司 B 私有云的服务器、CPU、计算能力等最强，是按时找到该随机数的参与者，因此它获得了本次账本的创建权，并创建了一个区块，假设为区块 x。此时，公司 B 私有云这个参与者需要将自己创建的区

<sup>①</sup> 比特币交易链采用约 10 分钟为限，其他基于区块链的应用也可以自行定义时间段。









新的交易与新的区块。一旦接收到新的交易或者新的区块均需首先验证它们的正确性，如果正确后再向自己的临近节点进行传播。如果接收到的新交易无效，则需立即抛弃，不再将它们转给临近节点，以免浪费计算资源。对于新交易的验证，根据基于区块链应用事先达成的各种验证协议来进行，比如交易的格式、交易的数据结构、格式的语法结构、输入输出、数字签名的正确性等。所有的新交易数据一旦验证通过后，节点会将这些交易数据放在一个交易池中。当节点确认了上一个区块以后，节点将按一定优先级次序从交易池中选出交易计算 Merkle 根。节点通过自己强大的算力（工作量证明）找到符合难度目标的随机数后，并在第一时间将新挖出的区块广播给其他节点，以便其他节点确认该区块，并将获得验证的新区块加入到原有的区块链中。

## 2.4 数据安全与隐私保护关键技术

无论是传统的应用还是基于区块链技术的应用，都面临数据安全与隐私保护问题。虽然区块链技术可以大大增强数据的安全性以及在一定程度上增加隐私保护的程度，但是仍面临不少挑战。本节简要介绍区块链技术中所面临的数据安全与隐私保护中所涉及的关键技术。

### 1. 时间戳

区块头里面必须包含一段时间戳信息。需要了解的是，基于“去中心”的设计思想，区块链的网络中一般没有一个中心化的时间服务器，因此各个节点的时间可能不一致，比特币区块链中的假设是各个节点的时间不超过 2 小时的偏离。因此，当我们谈到区块链的时间戳服务时，并不是指具体区块头里的时间戳信息，而是把区块链中的各区块生成的顺序作为一种广义的时间戳服务。这种区块顺序的时间戳服务可以防止双重支付，也可以提供一个溯源的作用。

### 2. 哈希函数

哈希函数，是一种从任何一种输入数据中创建小的数字“指纹”的方法。哈希函数把输入数据按一定的算法计算出来，生成固定长度的摘要，叫作哈希值的“指纹”。哈希值通常看上去像一些随机字母和数字组成的固定长度的字符串。哈希函数一般具有这样的特点：哈希值能和输入数据一一对应，给定输入数据，非常容易通过哈希函数计算得出固定长度的哈希值，但给定哈希值，要逆向计算出输入数据需要有天文数字的计算量。而且也没有办法找到两个相同的输入数据通过哈希计算得出同一个哈希值（密码学术语叫“碰撞”）。

安全哈希算法（Secure Hash Algorithm, SHA）是一个密码哈希函数家族。这一组函数是由美国国家安全局（NSA）设计，美国国家标准与技术研究院（NIST）发布。目前有 SHA-1、SHA-2 和 SHA-3 系列哈希标准。SHA-1 在 2005 年 2 月被中国密码学家王小云教授在理论上找到“碰撞”；2017 年 2 月，谷歌正式宣布实现对 SHA-1 的碰撞攻击。而早在



2011年,美国国家标准与技术研究院 NIST 已经推荐使用 SHA-2 系列哈希标准。SHA-256 就是 SHA-2 系列中的一个函数,是输出值为 256 位的哈希算法。到目前为止,还没有出现对 SHA-256 算法的有效攻击。它也是目前多数区块链平台所采用的哈希函数。区块链通常采用双哈希函数 (SHA-256),即将任意长度的原始数据经过 2 次 SHA-256 哈希运算后转换为长度为 256 位 (32 字节) 的二进制数字来统一存储和识别。

### 3. 数据加密

为了确保数据的传输安全,某些区块链应用需要对区块进行加密后再传输。另外,除了传输内容本身的加密外,为了安全传递密钥,也需要对密钥进行加密。数据加密算法主要有两大类,分别为对称加密算法和非对称加密算法。对称加密算法主要用于对区块链的交易和区块进行加密,其加密密钥和解密密钥使用同一把密钥。而非对称加密又包含两种:一种是公钥加密-私钥解密,就是我们平常所说的数字信封,其主要目的是用来安全传递密钥;另外一种是非对称加密-公钥解密,就是我们通常所说的数字签名,其目的主要用来作为签名使用,防止各种抵赖行为的发生。下面简要介绍数据加密的几种形态。

#### (1) 交易加密或者区块加密 (对称加密算法)

为了让区块链应用中所传输的交易或者区块安全保密,可以对交易信息或者区块信息采用对称加密算法进行加密,基本原理的简要描述如图 2-8 所示。

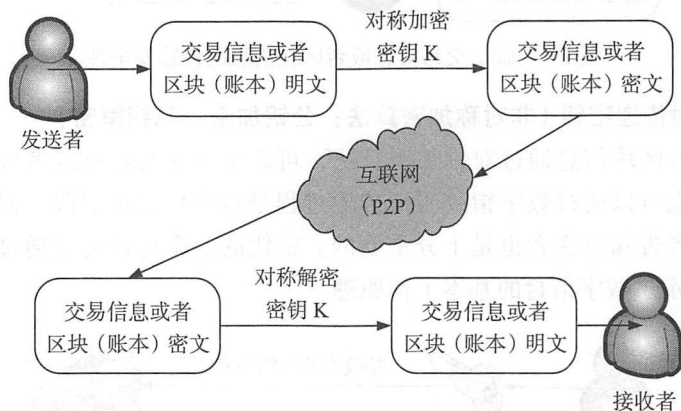


图 2-8 交易加密或者区块加密

从图 2-8 可以看出,为了实现交易信息或者区块 (账本) 的保密性传输,只要将传递的交易信息或者区块 (账本) 通过对称加密算法加密成密文传递,即可保障整个保密性的需要。为了验证信息是否在传输过程中被修改,可以通过图 2-9 所示的数字指纹方法进行验证。

从图 2-9 可以看出,为了保障交易信息或者区块信息的内容完整性,需要将传递的交易信息或者区块信息通过哈希计算得到一串哈希码  $h$ ,并将该哈希码  $h$  和传递的交易信息或者区块信息的密文一起发送给对方。对方收到信息并解密后,重新对解密后的明文进行哈





希计算得到一个新的哈希码  $h'$ 。然后对  $h$  和  $h'$  进行比较：如果  $h$  等于  $h'$ ，说明信息在传递的过程中没有被篡改，内容完整性得到保障；如果  $h$  不等于  $h'$ ，则说明信息在传递的过程中已经被篡改，内容完整性遭到破坏。

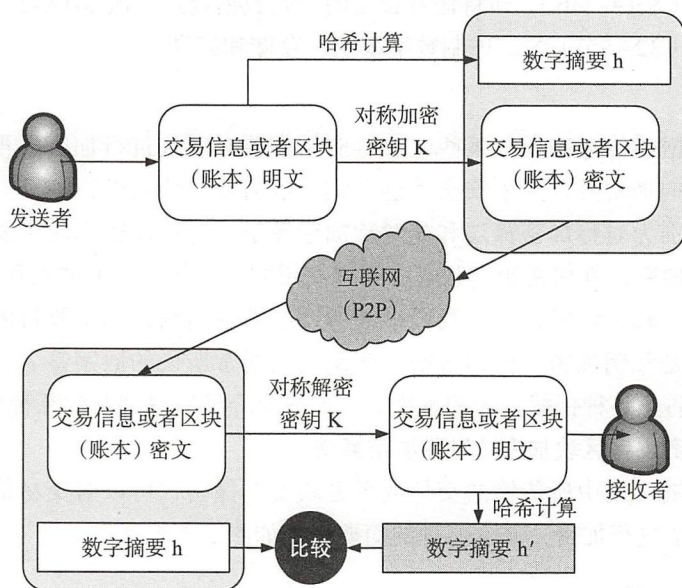


图 2-9 验证交易信息或者区块信息的传输安全性

## （2）数字信封传递密钥（非对称加密算法：公钥加密 - 私钥解密）

交易信息或者区块信息通过对称加密算法，可以实现交易信息或者区块信息的安全与保密传输，并且也可以通过数字指纹验证传输过程是否有信息被篡改。但是如何安全地将加密密钥从加密者告知解密者也是十分重要的。密钥的安全传递可以通过数字信封技术来实现，图 2-10 展示了数字信封的基本工作原理。

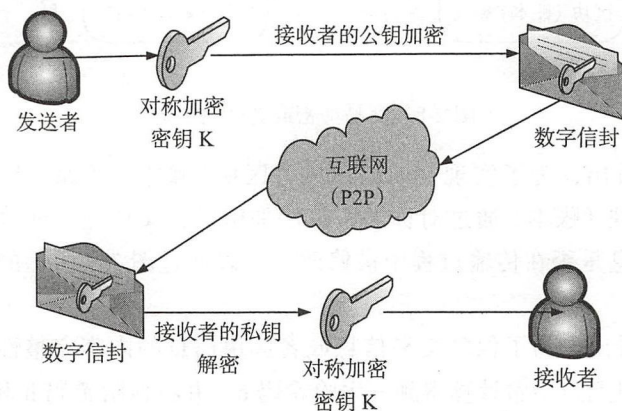


图 2-10 数字信封传递密钥





从图 2-10 可以看出, 为了实现交易信息或者区块信息的密钥安全传输保障, 需要将对称加密所用的对称加密密钥  $K$  通过使用接收者的公钥进行加密, 形成数字信封。该数字信封通过网络传递到接收方。接收方使用自己的私钥对该数字信封进行解密, 从而得到解密所需的对称解密密钥  $K$ 。因为在解密过程中, 只有接收者自己的私钥能够解开该数字信封, 任何其他人的钥匙都不能解开该信封, 所以数字信封可以保证整个密钥的传递安全性。

### (3) 数字签名防抵赖 (非对称加密算法: 私钥加密 - 公钥解密)

数字签名的主要目标是用来确认信息的发送者认可自己曾经的行为。类似传统的签名一样, 一旦某人签署了某份文件, 则表示其认可所签署文件的真实性, 并能证明为自己所签署。图 2-11 所示是数字签名的基本原理, 其核心就是采用非对称加密算法的私钥加密、公钥解密机制。

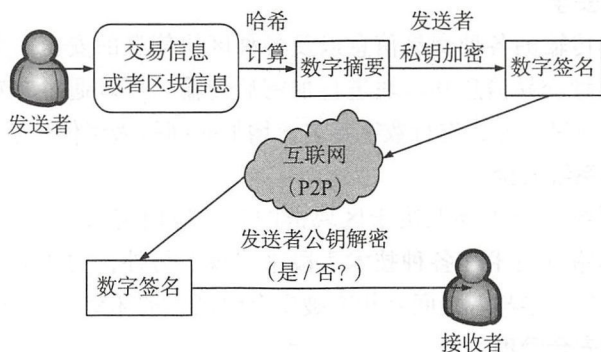


图 2-11 数字签名基本原理

从图 2-11 可以看出, 为了实现交易信息或者区块信息的来源可靠性保障, 需要将传递的交易信息或者区块信息通过哈希计算得到相应的数字摘要, 然后使用发送者的私钥进行加密, 得到相应的数字签名, 并将数字签名通过网络传递到接收方。如果接收方使用发送方的公钥能够解开该数字签名, 则证明该数字签名的确为该发送者所签署, 来源可靠; 如果接收方使用发送者的公钥不能解开该数字签名, 则证明该数字签名不是该发送者所签署, 来源不可靠。

## 4. 零知识证明

在数据安全层里面, 除了前面所描述的各种安全手段外, 有时候需要通过零知识证明来完成对区块链的隐私信息保护。零知识证明是由 Goldwasser 等人在 20 世纪 80 年代初提出的。它指的是证明者能够在不向验证者提供任何有用的信息的情况下, 使验证者相信某个论断是正确的。零知识证明实质上是一种涉及两方或更多方的协议, 即两方或更多方完成一项任务所需采取的一系列步骤。证明者向验证者证明并使其相信自己知道或拥有某一消息, 但证明过程不能向验证者泄漏任何关于被证明消息的信息。零知识证明不仅具有安全机制, 还在一定程度上实现了区块链应用的隐私保护作用。



## 5. 区块链安全体系

前面讲到了区块链安全管理中的一些基本的组件和关键技术，如哈希函数、数据加密算法、数字签名、零知识证明等。但是光靠这些基本的技术是不能形成区块链整个安全体系的。区块链应用是一个系统工程，需要有一个完整的安全体系，主要包括如下几个方面。

### (1) 物理网络环境安全

确保物理网络安全，主要包括电源安全、物理环境安全、防火墙安全、VPN 组网安全等。例如，对于一些敏感重要的区块链应用可以组建私有链，所有的数据传输均在公司内部的 VPN 网络中进行传输，从而进一步提升物理网络的安全性。如果是一些公司组成的联盟，也可以在联盟内部组建一个联盟 VPN 网络，来确保物理网络安全。

### (2) 区块链数据安全

主要确保区块链传输的各种交易信息以及各种区块信息的安全。为了确保信息传输安全，可以对传输的各种交易信息和区块进行加密后传输。可以通过非对称加密的方式协商密钥的传递，也可以通过该方式进行数字签名，增加时间戳来确保信息的时间维度等。

### (3) 区块链应用系统安全

区块链的应用系统安全主要取决于区块链的各种应用需求，可以通过身份认证技术、权限访问技术、访问审计技术等各种技术手段来实现。另外，构建应用系统时，必须制定各种交易规则，防止各个参与者之间互相突破安全限制、破坏交易规则等。

### (4) 区块链密钥安全管理

区块链环境下，一般依靠钱包来管理密钥。密钥，特别是私钥，在区块链应用中直接决定资产的归属权。因此私钥保护是区块链安全的重中之重。如何利用多重签名技术保证资产安全，如何安全地备份钱包、恢复钱包，建立一整套钱包管理流程也是一个需要考虑的课题。

## 6. 隐私保护机制

目前很多公有区块链上传输和存储的数据都没有经过隐私处理，仅仅采用简单的匿名方法对区块链上的参与者进行一定程度上的隐私保护。但是随着区块链技术的应用越来越广，如何保证用户的隐私安全显得尤其重要。目前有一些新的技术来保护隐私安全，主要有同态加密技术、零知识证明以及环签名等。未来需要形成一套更为有效的隐私保护机制，以满足不同需求的区块链应用的需要。

区块链隐私保护算法和传统的云环境下的隐私保护一样，主要涉及数据生命周期中的不同阶段所采用的不同算法。主要有如下几个阶段。

### (1) 交易信息或区块创建隐私保护算法

交易信息或区块在创建的时候，其中最需要保护隐私的是对创建者进行匿名，让交易信息或区块链在后续的传播过程中不知道该交易信息或区块链是谁创建的。因此本阶段的交易信息或区块链隐私保护算法主要需要采用各种匿名算法，如最典型的 k-匿名等。





### (2) 交易信息或区块链存储隐私保护算法

交易信息或区块创建好后, 将保存在互联网环境(甚至云环境)下。确保所保存的交易信息或区块的内容不被发现, 可以通过各种算法来实现, 其中最为典型的是采用加密算法(包括云环境下的同态加密), 让所有人看不见原文, 只能看见密文, 从而起到保护隐私的作用。当然, 有时候不一定要将所有的全文进行加密, 只要从明文中将那些敏感的信息通过隐私抽取方法抽取出来, 将隐私信息通过加密方法单独存储起来, 也可以达到实现隐私保护的目的。

图 2-12 展示了大数据环境下的交易信息或区块链存储过程中的隐私保护模型。大数据在经过隐私信息提取后, 将分解成共享信息、隐私信息位置语义映射表及隐私信息 3 大部分。其中共享信息将存储在公共云或者共享存储集群中, 以供数据使用者共享使用。隐私信息位置语义映射表记录了大数据的隐私信息在原始大数据中的具体位置的一个映射表, 为将来的数据融合提供基础。隐私信息将经过加密处理后存储到数据库中进行安全保存。另外在整个大数据的隐私处理过程中, 所有大数据操作过程作为隐私信息也将被提取, 进行保密处理并安全存储。大数据的提供者可以对隐私信息和共享大数据进行输入融合, 还原原始的大数据信息。另外, 大数据的提供者还可以针对各种操作过程, 进行大数据溯源, 确保大数据在每个操作中都有据可查, 进一步确保整个安全和隐私得到保护。一旦出现隐私泄露, 也为法律取证提供证据支撑。

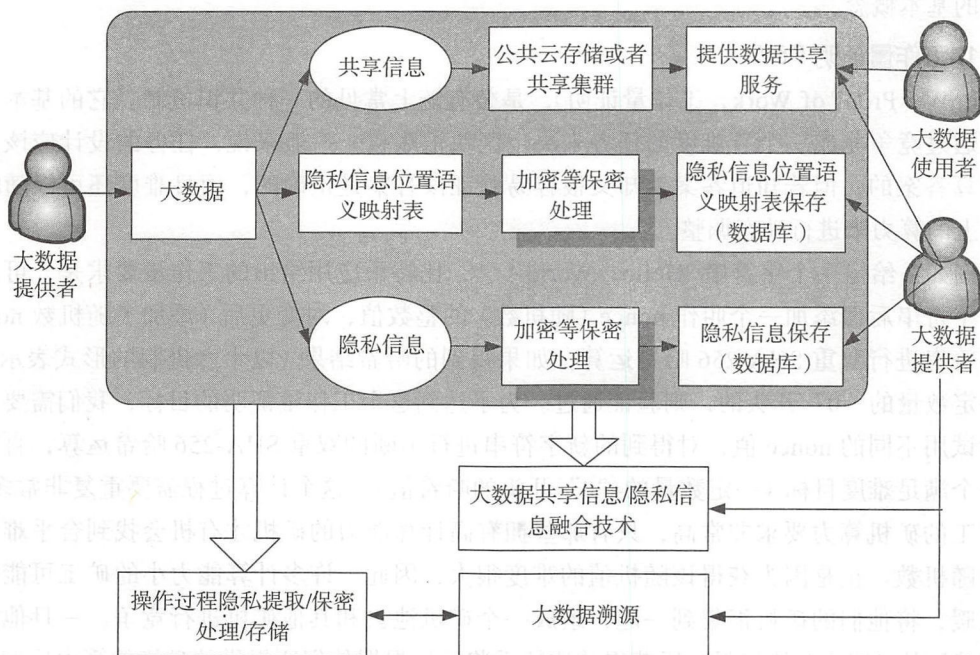


图 2-12 大数据环境下的交易信息或区块链信息隐私保护模型

### (3) 交易信息或区块链数据挖掘过程中隐私保护算法

隐私信息包括两大类, 一类是直接隐私, 另一类是间接隐私。直接隐私是指大数据中





直接包含的隐私信息。如医疗区块链应用中的各种医疗病历中的患者姓名、年龄、出生地点、病名及其工作单位等。这类隐私是直接可以通过查看医疗区块链的交易信息或者医疗区块直接获取的。间接隐私是指不能从医疗区块链应用中的医疗交易信息或者医疗区块所构成的大数据中直接获取的隐私信息，是需要通过一定的算法或者方法，通过对医疗区块链的各种大数据进行各种数据挖掘之后得出的隐私信息。例如，通过各种关联规则的方法可以挖掘出相应的隐私信息。而这里主要是指各种间接隐私的保护算法，主要有数据变换算法、数据隐藏算法等。

#### （4）交易信息或区块链在用户使用过程中的隐私保护算法

所有的交易信息或者区块信息都要被区块链应用中的各种参与者所访问或者使用，最后一个环节就是用户使用过程中的隐私保护算法。这种算法主要是对用户角色和权限进行控制，确保用户的访问范围。

## 2.5 共识层关键技术

基于区块链技术的各种应用与其他应用有一个明显的区别，即区块链应用需要依靠区块链底层平台提供的共识机制来确保信任和应用的各种状态确认。下面简单描述不同共识机制的基本概念。

### 1. 工作量证明 PoW

PoW（Proof of Work，工作量证明），是公有链上常见的一种共识机制。它的基本思想是，通过竞争完成一个有难度的任务来决定区块记账权。一般来说，任务的设计应该是很难计算答案的，但若知道答案，却又很容易验证该答案是正确的，而且难度还可以随时根据网上的算力来进行动态调整。

例如：给定一个字符串 "Hello, World！"，比特币应用给出的工作量要求是，可以在这个字符串后面添加一个叫作 nonce（随机数）的整数值，对变更后（添加了随机数 nonce）的字符串进行双重 SHA-256 哈希运算，如果得到的哈希结果（以十六进制的形式表示）是以一定数量的“0”开头的，则验证通过。为了达到这个工作量证明的目标，我们需要不停地尝试用不同的 nonce 值，对得到的新字符串进行不断的双重 SHA-256 哈希运算，直到找到一个满足难度目标（一定数量的“0”开头的哈希值）。这个计算过程需要重复非常多次，对矿工的矿机算力要求非常高，只有那些拥有高计算能力的矿机才有机会找到合乎难度目标的随机数。正是因为获得该随机值的难度很大，因此，许多计算能力小的矿工可能会抱团取暖，将他们的矿机汇集到一起，组成一个矿机池去和其他矿机进行竞争。一旦他们获得创建区块（账本）的权限，所获得的比特币奖励将根据他们所提供的矿机的算力比例进行分配。

### 2. 权益证明 PoS

比特币的 PoW（工作量证明）通过算力竞赛来形成对记账权的共识，其工作量也没有



其他价值。因此这种机制其实带来了巨大的资源浪费。为了克服 PoW 造成的巨大浪费，后来发明了一种 PoS（权益证明）的共识机制。

PoS 是点点币（PPC）最早所采用的一种共识机制。PoS 机制主要是根据参与者手中持有代币的多少和时间长短（币龄）来决定获得出块记账权的概率，持的币龄越多，被选中出块记账的机会就越大，获得的奖励就越多。与 PoW 相比，PoS 的优点是能大幅提升共识效率，降低共识成本，减少算力浪费。PoS 的缺点是安全性比较差，节点可以低成本地分叉作恶，造成 PoS 公平性先天不足。

### 3. 股份授权证明（DPoS）

PoS 共识机制与 PoW 一样，每个节点都可以创建区块，这样参与共识的节点比较多，效率也相应比较差。与 PoS 不同的是，DPoS 是由被社区选举的可信账户（受托人）来创建区块。为了成为正式受托人，用户要去社区拉票，获得足够多其他用户的信任。用户根据自己持有的加密货币数量占总量的百分比来投票。DPoS 机制有点类似于股份制公司，广大股民通过选择自己最信任的股东代表来参加公司管理。最后得票率最高的一定数量的参与者获得创建区块以及完成交易信息或者区块信息验证的代理权。

相比于 PoW 共识机制和 PoS 机制，DPoS 机制只需要更少的区块链应用的参与者来完成区块的创建以及各种交易信息的验证，从而大大提升了传播速度（因为不需要在全部的区块链应用参与者中进行全网传播，只需要在这些代理之间实施传播即可），同时也节省了大量的算力资源，节省了大量的能源消耗。但是 DPoS 机制是建立在社区选举的基础之上的，选举的可靠性和安全性将直接制约 DPoS 机制的有效性，也将带来一些安全隐患。

### 4. 拜占庭容错共识机制

拜占庭容错共识机制是一个古老的容错机制，例如 PBFT 共识机制。拜占庭位于今土耳其的伊斯坦布尔，是当时东罗马帝国的首都。由于当时拜占庭罗马帝国国土辽阔，出于防御目的，每个军队都分隔很远，将军与将军之间只能靠信差传消息。在战争的时候，拜占庭军队内所有将军和副官必须达成共识，确定有赢的机会才去攻打敌人的阵营。但是，在军队内有可能存在叛徒和敌军的间谍，既左右将军们的决定又扰乱整体军队的秩序。在确认共识时，结果并不代表大多数人的意见。在已知有成员谋反的情况下，忠诚的将军在不受叛徒影响的情况下如何达成一致，拜占庭问题就此形成。

### 5. 其他

除了前面讲述的几种共识机制外，各种区块链应用不断产生一些新的共识机制来达到某种共识。随着应用需求的不断扩大，会产生越来越多合适的其他共识机制。

## 2.6 区块链技术标准

由于区块链技术从 2015 年才引起关注，2016 年开始逐渐火爆，但尚未得到足够的重视，



因此,目前国内外还没有形成统一的技术标准。为了促进区块链技术的标准化和引导区块链技术的产业化,尤其是区块链应用的跨境国际化,需要形成统一的技术标准。

2016年10月18日,在工业和信息化部信息化和软件服务业司以及国标委指导下,中国区块链技术和产业发展论坛编写的《中国区块链技术和应用发展白皮书(2016)》正式亮相,区块链技术迎来了第一个官方指导文件。

2016年12月31日,贵阳市政府正式提出“主权区块链”与“秩序互联网”等创新理论,梳理了贵阳市探索区块链技术对政务、民生、商务发展应用的总体设计蓝图。

2017年5月16日,在杭州国际博览中心举行的区块链技术应用峰会暨首届中国区块链开发大赛成果发布会上,首个区块链标准《区块链:参考架构》正式发布,推动了区块链的产业化进程。

区块链作为一种新兴的应用模式,在金融服务、供应链管理、文化娱乐、智能制造、社会公益和教育就业等领域有着广泛的应用价值。近几年来,区块链技术和应用正经历快速发展的过程。与此同时,国内国际上区块链领域的标准仍属空白,行业发展碎片化,行业应用存在一定的盲目性,不利于区块链的应用落地和技术发展。区块链的标准化有助于统一对区块链的认识,规范和指导区块链在各行业的应用,以及促进解决区块链的关键技术问题,对于区块链产业生态发展意义重大。

## 2.7 小结

本章介绍了区块链的基本概念和区块链的基本技术组件,覆盖了区块链核心技术中P2P网络、密码学、共识机制、智能合约等各技术组件的基本概念。首先,以比特币为例说明了区块链技术的由来以及比特币的基本运行机制。然后给出了区块链研究的基本架构,分析了其基本关键技术,包括:区块链区块的基本结构、链式结构、Merkle树等存储机制,区块链的传播和验证等网络通信机制,数字签名技术、零知识证明等区块链的安全和隐私保护机制,工作量证明、权益证明等区块链的各种共识机制和算法,发行机制、分配机制、智能合约、可编程资产等应用层组件,区块链的各种应用,区块链的标准以及区块链与现代技术的融合等。本章的目的是让读者对区块链技术有一个初步的认识。后续章节将针对具体的区块链技术和区块链平台进行详细阐述。

### 1. 习题

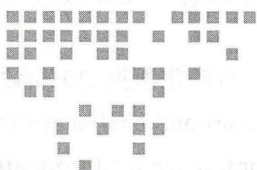
- [1] 区块链技术模型是什么?
- [2] 什么是Merkle树?并简述其基本原理。
- [3] 简述区块链环境下的数字签名原理。
- [4] 区块链的共识机制主要有哪些?

### 2. 参考资料

- [1] Nakamoto.Bitcoin: A Peer-to-Peer Electronic Cash System. Satoshi.2008.



- [2] 蔡维德, 郁莲, 王荣, 等. 基于区块链的应用系统开发方法研究 [J]. 软件学报, 2017, 28(06):1474-1487.
- [3] 袁勇, 王飞跃. 区块链技术发展现状与展望 [J]. 自动化学报, 2016, 42(04):481-494.
- [4] N. Chalaemwongwan, W. Kurutach. State of the art and challenges facing consensus protocols on blockchain. 2018 International Conference on Information Networking (ICOIN), Chiang Mai, Thailand, 2018, pp. 957-962.
- [5] Yu-Long Gao, Xiu-Bo Chen, Yu-Ling Chen, etc. A Secure Cryptocurrency Scheme Based on Post-Quantum Blockchain. Special Section on The Internet of Energy: Architectures, Cyber Security, and Applications (Part II). 2018(6), pp:27205 - 27213.
- [6] P. Tasatanattakool, C. Techapanupreeda. Blockchain: Challenges and applications. 2018 International Conference on Information Networking (ICOIN), Chiang Mai, Thailand, 2018, pp. 473-475.



## Chapter 3

## 第 3 章

## 区块链架构

作者：庄鹏

从前面两章我们可以看到，与其说区块链是一种技术创新，不如说它是一种架构创新。它将已有的分布式计算、密码学、网络技术和虚拟机技术有机结合，形成一个全新的架构。作为一种架构创新，区块链以其去中心化的基因，给传统中心化的 IT 架构带来颠覆性的变革，极大地降低了传统中心化系统建立信任和维护信任的成本，这不仅包括金钱上的成本，还包括时间上的成本。本章将带你一起探讨区块链的本质，给出区块链通用和跨链的架构模型、数据模型以及能力模型。

### 3.1 区块链架构和传统 IT 架构的异同

我们可以把世界上很多系统看成有限状态机（Finite State Machine 或 Automata）。在外界的作用下，这些状态机随时间变化不停进行着状态演变。举一个简单的例子，ATM 就是一个有限状态机，它有等待输入指令、银行请求等待、出错提示、出钞等状态。外界输入的指令不同，其状态也会发生转变。

传统的 ATM 机都要和银行的中心化账户系统对接，受中心化的银行系统控制。我们可以假设有一个区块链 ATM 平台，以对比区块链 ATM 和传统 ATM 架构的异同。传统的 ATM 系统是一个中心化的分布式系统，其中心是银行的核心系统，能够并行执行任务，允许很多人同时在 ATM 上操作，只是在关联交易的账户之间的操作上需要在银行的账户系统做串行处理。比如张三在 ATM 上将其 A 账户的钱转给他的 B 账户，那么在张三 A 账户中减掉一笔资金并在其 B 账户增加相应一笔资金的操作就必须串行执行。ATM 中各账户的状态需要中心化银行的核心系统来确定。

我们再来看区块链 ATM 系统，它也是一个分布式系统，但没有中心节点，也没有中心化的核心系统，每个 ATM 都是独立的有限状态机，都有独立的一套银行核心系统，支持并行执行任务，可以很多人同时在不同的 ATM 上做交易。但每笔交易需要广播给网络中所有的 ATM。每隔一个固定时间，所有的区块链 ATM 需要按照一个共识算法来确认账户的状态，并防止同一笔钱被花掉 2 次。这样，在共识阶段，相当于每个区块链 ATM 都要执行所有的交易。这样看起来，去中心化的区块链 ATM 的效率要比中心化的 ATM 低一个网络节点数倍数。因为传统 ATM 只执行在本 ATM 上操作的交易，只有中心化的银行核心系统执行所有交易。而区块链 ATM 上每个 ATM 都要执行在其他 ATM 上操作的交易。这个就是中心化的分布式系统和去中心化的区块链分布式系统的根本不同。

另一个更容易理解的例子是云计算。云计算的虚拟化技术可以把一台物理服务器虚拟成多个虚拟机，然后把一个任务拆分，在多个虚拟机上并行执行，最后把各个虚拟机上执行的结果传回到云计算的中心管理节点，串行组合成最后结果。这是个“一变多”的并行。而完全去中心的区块链平台，则是每个任务都在每个节点执行，结果在全网做共识，是一个“多变一”的串行执行。由此可见，从架构来说，去中心化的区块链架构比中心化的分布式系统在效率上要低很多。

这个效率的代价换取的是低成本的信任构建和信任维护，换取客观公正性、公平性和透明性，同时也换取整个系统的健壮性和安全性。

## 3.2 区块链架构模型

本节提出一个区块链架构模型，并探讨区块链的两种典型数据模型，还给出区块链的能力模型，以期引导读者从多个维度对区块链有比较深入的理解和领悟。

### 3.2.1 区块链系统的共性

区块链对现实世界的建模不同于传统 IT 系统对现实世界的建模。区块链系统所要建模的是现实世界里不同主体通过各种方法所达成一致的确定性信息状态，即价值、合约、流程。其不同点就在于区块链使用一种集体参与、不可篡改的方法建模不同主体共同确认的信息状态，并记录修改状态的变化。这些被建模的对象可被各分布式节点所验证，而且不同的节点验证的结果是相同的。传统的 IT 系统通过中心化的主控系统建模对象的信息状态变化，需要假设存在对中心化主控系统的无条件信任。区块链的参与者是不同的行为主体，如何在不同的行为主体之间进行协调呢？一方面是建立各节点需要遵守的区块链技术协议，另一方面是建立一个共识机制，保证即使有少数节点不遵守协议规则，也能保证所有正常的节点形成对系统状态的一致性确认。区块链有以下几个共性。

#### (1) 共识 (consensus) 机制

你看到的就是我在看到的，我知道我看到的就是你看到的，各个分布节点使用相同的共



共识算法可以得到对建模对象相同的认识，也可以确信其他人的认识和自己是一致的。共识是保证所有节点的分布式账本的规范性和一致性的机制与标准。公有链由于不能有任何信任假设，唯一能信任的就是数学，采用基于哈希算法的工作量证明 PoW，虽然消耗大量的能源，但也保证了无信任情况下的网络安全；如果引入股权抵押策略，公有链可以基于更加环境友好的 BFT 算法实现基于投票的共识。因此公有链需要引入内生的代币，通过代币结合经济手段（正向激励和反向惩罚）进行公有链的运行和治理，如 Cosmos<sup>①</sup>、Polkadot<sup>②</sup>。许可链（Permissioned Chain）<sup>③</sup>由于是在受控环境下运行的，对参与主体有严格控制，任何违法行为都可以被有效惩处，所以一般不发行代币进行经济激励，但它可以采用 BFT 投票算法实现参与主体间的共识，如 Fabric 0.6，也可以采用更加高效的验证和规范化分离的架构，验证甚至只在交易相关方的节点进行，由完全不需要知道业务细节的技术节点实现唯一性共识，如 Fabric 1.0 的时序节点进行全时序排序保证交易的唯一性。

### （2）状态可验证性（validity）

系统中所有当前的状态都是确定的，任何比特币的未花费交易（UTXO）、以太坊的账户余额和合约账户状态、超级账本 Fabric 的合约状态、R3 Corda 的状态资产，都可以被各自既定的确定性交易或流程规则所验证。确定性的交易、服务、流程规则，任何时间在任何验证节点上运行，相同的输入状态可以得到相同的输出状态。那些依赖于时间、随机数的规则都不能作为合约的规则；那些需要依赖于现实世界事件的规则都需要以预言机（Oracle）服务的方式确定性地提供。确定性验证一般在沙盒中运行，以减少外界信息的干扰，如比特币使用脚本栈、以太坊使用 EVM、Fabric 使用 DockerVM 或 JVM、Corda 使用 JVM、EOS 和 Polkadot 使用 WebAssembly 虚拟机。对于全节点，需要进行交易的实际执行验证，而对于轻节点，可以只基于区块头做简化验证，如比特币只需基于部分 Merkle 树路径节点进行简化支付验证（Simple Pay Validation, SPV）。对于交易执行时将交易日志记录到收据中的交易，如以太坊，存在性验证可以通过更高效的布隆过滤器实现。对于跨链的 Cosmos 和 Polkadot，更需要这种简化验证以提高跨链交互效率。

### （3）唯一性（uniqueness）

任何状态资产的消费只能发生一次，即区块链的防“双花”安全性。

❑ 比特币相同 TxID 的未花费交易不会存在于同一个 block 中，也不会存在于连续的 block 中，如果出现在分叉 block 中，长链最终会胜出，这样可确保消费的唯一性。

❑ 以太坊相当于一台世界计算机，按顺序执行合约代码，类似 Merkle 的 Patricia 树组

---

① <https://cosmos.network/whitepaper>。

② <https://polkadot.network/Polkadot-lightpaper.pdf>。

③ 许可链参与共识的主体必须经过准许才能加入网络、发起交易，这区别于公有链，公有链不限制任何主体的节点加入网络，也不限制任何主体向网络发起交易。许可链典型的准许方式是通过证书证明自己的身份，证书必须经过区块链认可的 CA 签发才可以组网、发起交易，否则许可链网络会丢弃不合法证书发起的组网请求和交易请求。一般来说，包含联盟链和私有链。

织起了所有账户的余额和合约账户状态，每个账户会记录交易执行顺序号，确保任何时刻交易的唯一性和各账户余额与合约状态的唯一性，也采用长链胜出的最终确认法则。

- ❑ 超级账本 Fabric 的合约状态基于每次交易执行产生的读写状态版本号集合确定，交易导致读写状态版本号集合的变化是确定的和唯一的，通过全局时序器保证交易的唯一性。
- ❑ R3 Corda 采用类似比特币的 UTXO 模型，它属于许可链，通过让 notary 服务公证人签名的方式确保状态资产的唯一花费。
- ❑ EOS 对于发往每一个账户的消息分配一个顺序号，以保证消息处理的完备性和唯一有序性。
- ❑ Cosmos 在执行跨链通信时，为跨链载体 packet 指定唯一编号以保证 packet 在各个 Zone 链和 Hub 链的唯一性。
- ❑ Polkadot 的任何一个平行链 egress（出口）交易队列在同一个区块时间内被中继链转移到目的平行链的 ingress（入口）队列中，所有的出口 egress 队列和入口 ingress 队列的消息进入与取出后都进行 Merkle 树哈希计算，保证了消费的唯一性。

#### （4）不可篡改（immutability）

密码学方式组织的区块链确保了已经提交到区块中的交易不可更改，否则会导致全盘修改。比特币、以太坊、超级账本 Fabric、EOS 都是采用 Merkle 树 + 区块哈希指针数据结构串接交易记录和状态记录，以达到交易和状态不可篡改的目的。比特币使用 BerkeleyDB 文件数据库存放区块文件，并使用 LevelDB 进行区块索引和记录 UTXO 交易，以太坊使用 LevelDB 记录 Patricia 树，超级账本 Fabric 使用 LevelDB 持久化合约状态。R3 Corda 使用关系型数据库 H2 存放交易数据，但是其存放的交易数据是不可修改的，交易数据由 inputs、outputs、commands、attachments 构成，采用传统的 Merkle 树构建出的 Merkle 树哈希根作为交易 ID 记录在关系型数据库中。Cosmos 和 Polkadot 都同时存在多个分链和总链的多层链结构，分链的交易或区块信息也会成为总链区块的组成部分。

#### （5）基于公 / 私钥的认证（authentication）

不同于传统系统的管理员授权权限模式，区块链完全采用基于密码学的公 / 私钥方式通过签名来认证个人身份，所有的资产消费和验证也都围绕公 / 私钥进行，公有链中如比特币、以太坊、EOS 只需要公 / 私钥，不需要 CA 证书。而许可链，如 Fabric、Corda 一般是基于 PKI 体系完成个人或节点证书的 CA 中心证书签名，交易权限的控制是基于个人或节点证书对交易私钥签名的验证，如验证交易发起人的身份和权限。

#### （6）区块链治理（governance）

一个区块链就是一个包含不同参与主体的生态体系，一方面需要维护整个区块链生态按计划演进和健康发展，一方面会面临许多公共决策和纠纷解决，如里程碑代码分叉、Bug 修复、恶意行为惩罚、运行参数调整等。同现实世界一样，区块链需要建立一套治理机制



做保障。比特币和以太坊都有自己的核心开发组织和基金会，然而起初并没有考虑链上和链下治理机制的必要性，导致了比特币协议升级艰难，以太坊出现分叉链的后果。后续的区块链 EOS、Cosmos 和 Polkadot 都意识到了这一点，开始制定区块链宪法，并提供一定的链上治理机制和链下治理流程。

上述各个方面的共同点，使得这种不可篡改的、在不同主体间的共同认知拥有了内在的价值，并且有一套规则和治理机制维护和保障主体所拥有的价值，所以建模状态都存在内在的信任价值也是所有区块链的共同之处。

### 3.2.2 区块链的差异性

不同代际、不同使用场景的区块链，会有不同的架构决策。在公有环境下运行的公有链和在联盟环境下运行的许可链，会因实际运行环境、参与主体、监管需求、非功能性需求的不同，设计出不同的技术架构、共识机制、隐私保护机制、权限控制机制、经济模型和治理架构。对于不同的区块链代际、区块链货币、区块链金融和区块链社会，因建模对象、业务自由度、价值交换自由度、法律法规渗入程度、自动化程度的不同，也会在系统架构、运行机制和平台策略上有不同的设计要求。以下列出了一些区块链的主要不同之处。

#### (1) 共识主体不同

公有链，如比特币、以太坊的共识主体可以是任何人，或者人的程序代理和人工智能代理，这种公有链的安全建立在一种完全随机性的数学运算工作量上，如 PoW。EOS、Cosmos、Polkadot 共识主体是每轮次通过股权证明投票（或委托投票）选举出来的验证人，采用选举算法投票共识的公有链的安全建立在股权质押机制上。诚实的验证人获得正向经济激励，失误或恶意的验证人受到不同程度的惩罚。超级账本 Fabric 和 R3 Corda 的共识主体是准许参与的主体，是一种基于许可的区块链，由于对参与主体完全可控，可以基于预先设定的惩罚规则和对运行环境的完全控制确保网络安全。如：Fabric 建立了基于 PKI 体系的多 CA 中心认证机制，确保参与的节点和用户发起的交易都是受控的；Corda 网络所有节点的证书都收到同一个根 CA 签名以允许访问网络。

#### (2) 建模对象范围不同

建模对象需要能在各参与方形成共识。比特币建模对象最简单仅是 BTC 金额；以太坊建模的是各个账户余额和合约账户状态（轻状态）；超级账本 Fabric 建模对象是合约代码状态（轻状态）；R3 Corda 建模对象是金融资产状态（UTXO 状态，如票据）；EOS 建模的是一种应用状态；Cosmos 和 Polkadot 除了建模各个分链和总链的状态，还建模跨链的交互消息。目前的区块链都只是对合约状态的建模。如果要想实现区块链社会的颠覆性创新，未来的区块链还需要拥有对业务流程状态进行建模的能力，也就是要有对状态机进行共识建模的能力，并且还要能实现链内及跨链共识流程驱动的合约服务的调用。当前合约都是以合约代码方式出现的，如果要想实现流程的共识建模，就需要把目前编写的合约代码提升为编写标准化的合约服务规约。合约服务是人类可以直接理解的语义层面的共识描述，是对



业务合约的逻辑抽象和形式化语义描述。

### (3) 建模对象操控能力不同

比特币仅仅是在脚本栈中运行指定的脚本验证和执行状态转换,不能实现复杂的业务处理逻辑。以太坊、R3 Corda、Fabric、EOS、Cosmos 和 Polkadot,都引入了合约或链上代码的概念,采用高级语言进行状态建模和状态转换,在隔离的沙箱中运行。隔离的沙箱可以是 EVM、DockerVM、JVM、WebAssembly,它们都属于图灵完备型。未来的区块链需要对业务流程和合约服务进行建模和共识验证,就需要有一个支持流程引擎的虚拟机和支持合约服务注册与路由的虚拟机提供计算支持。

### (4) 法律法规支持程度不同

比特币和一些衍生币由于只是对货币的建模,没有复杂的合约逻辑,在法律法规支持上也没有特别的要求。而以太坊、Fabric、Corda,一旦实现的合约逻辑涉及现实世界的多个法律参与主体,就面临着同现实世界合同类似的法律问题。所谓的“code is law”(代码即法律)是不符合人类自身利益的,特别是随着人工智能技术的发展,这一点更加重要。只有坚持虚拟世界的运行规则最终要体现和符合现实世界人类的法理,人类才能是虚拟世界的主宰,否则随着人工智能的渗入和机器人在现实世界的普及,人类可能会面临虚拟和现实世界的全面失控。合法合规自动化检查可以是链上治理的一部分功能,这样就能在虚拟世界中为各式各样的合约服务和合约流程设定运行规则、行为约束和惩罚标准,以虚拟世界的法律体系为主体,结合现实世界的法律体系,共同维护虚拟世界的公平正义以及人类对虚拟世界的主导权。

### (5) 隐私和匿名性不同

比特币的出现带来了区别于传统 IT 的隐私模型,个人的隐私由个人做主,隐私信息和交易自主权不再被第三方机构所掌控。公有链中会逐渐采用零知识证明 zkSNARK<sup>①</sup>这一密码学最新研究成果,实现无须知道交易细节就可验证的交易匿名性,以保护用户和交易的隐私。如 Zcash 在比特币基础上提升隐私和匿名性,以太坊决定引入 zkSNARK 隐藏交易内容,Polkadot 也引入 zkSNARK 提供跨链证明。联盟链,如 Fabric 和 Corda,由于准入机制和网络可控性,在交易验证上不像公有链有那么高的分布性要求,可以允许交易在交易相关方进行验证,这样即使是加密的数据也不会分享给无关的网络参与方。

### (6) 交易延迟确认的不同

交易确认有两种,一种是立即确认,一种是最终确认。最终确认实际上是一种概率确认,如等待多少个区块后,交易所属区块就不可能被推翻。这与采用什么样的共识算法密切相关。如果采用 PoW 算法,交易都是最终确认,如比特币和 Zcash 需要 6 个区块,以太坊需要 120 个区块 PoW 算法采用长链胜出原则,在若干个区块后才能判断出哪个是长链,所以一般确认的时间较长。如果采用 BFT 算法,全都采用 2/3+1(绝大多数)投票规则,投

<sup>①</sup> 推荐阅读 <https://z.cash/technology/zksnarks.html>。

票结束就能在所有投票人之间形成共识，所以可以实现立即确认 Cosmos。但如果 BFT 算法投票中包括非绝大多数情况，如 Polkadot 中 1/3 数据可用性投票，就有可能存在分叉，最终确认区块数同平行链的个数成正比。如果采用 DPoS 的 EOS，由于有 21 个验证节点，可能存在恶意节点区块的情形，需要 15 个区块生产者（2/3）产块后最终确认。Fabric1.0（时序器采用 Kafka 时）、Corda（Notary 采用 RAFT 或 BFT）也是立即交易确认。

### （7）性能和可扩展性不同

不同风险特性、不同隐私要求、不同性能要求的业务应用类型，混在一起执行，共同维护在一个账本中，以单一角色的客户端运行，导致性能无法提升和扩展性差，无法满足吞吐量和响应时间需求。后来区块链把不需要混在一起的角色进行分离，按照隐私团体、应用类型分别建链，把合约执行、共识算法作为模块化拆分出来以插件化方式提供，引入交易并行验证调度器来并行验证无依赖关系的交易，采用标准的接口实现应用层与共识层分离，采用统一的报文或协议实现不同链间的交互和安全性。应用架构层面的纵向分层和横向层面的节点角色分离，并结合容器化服务集群、分布式可扩展架构，实现高扩展性、高性能的要求。如运行 Kafka 时序器的 Fabric1.0，就分成了时序节点（Orderer）、背书节点（Endorser）、确认节点（Committer）消息中间件节点（Kafka）、分布式注册节点（Zookeeper）、CA 节点（CA）、DApp 节点。而 Polkadot 也存在收集人、验证人、可用性保证人、钓鱼人、委托人等多种角色节点。

## 3.3 区块链的参考模型

### 3.3.1 一个参考架构

2017 年 5 月 16 日上午，由工信部指导下的中国区块链技术和产业发展论坛正式发布了首个区块链标准《区块链参考架构》。《区块链参考架构》从用户视图、功能视图给出了区块链参考架构。用户视图包含角色、活动以及角色之间的关系，功能视图包含功能组件以及功能组件之间的关系，另外还给出了用户视图和功能视图中各模块之间的对应关系。这里仅引用其功能视图，如图 3-1 所示。

功能视图通过“四横四纵”的层级结构（包括用户层、服务层、核心层、基础层，以及包含开发、运营、安全、监管和审计的跨层功能）描述了区块链系统的典型功能组件。同时规定了区块链的 7 个共同关注点，包括模块化、性能、互操作、数据一致性、安全和隐私、经济合理以及安全可信。《区块链参考架构》还总结了区块链的典型特征，包括分布式对等、数据块链式、不可伪造和防篡改、透明可信和高可靠性；定义了区块链的 3 种部署模式，即公有链、联盟链和私有链；规定了区块链服务能力类型，包括基础设施、数据和应用服务能力，以及基于这些服务能力的区块链服务类别。

标准、实现、治理是任何技术成功应用的三要素，区块链技术和应用随着时间推移也



必然会在这三个要素上越来越完善。在标准形成之前，架构、业务和技术框架显得尤为重  
要，它可以指引着我们前进的方向，是标准形成的基础。

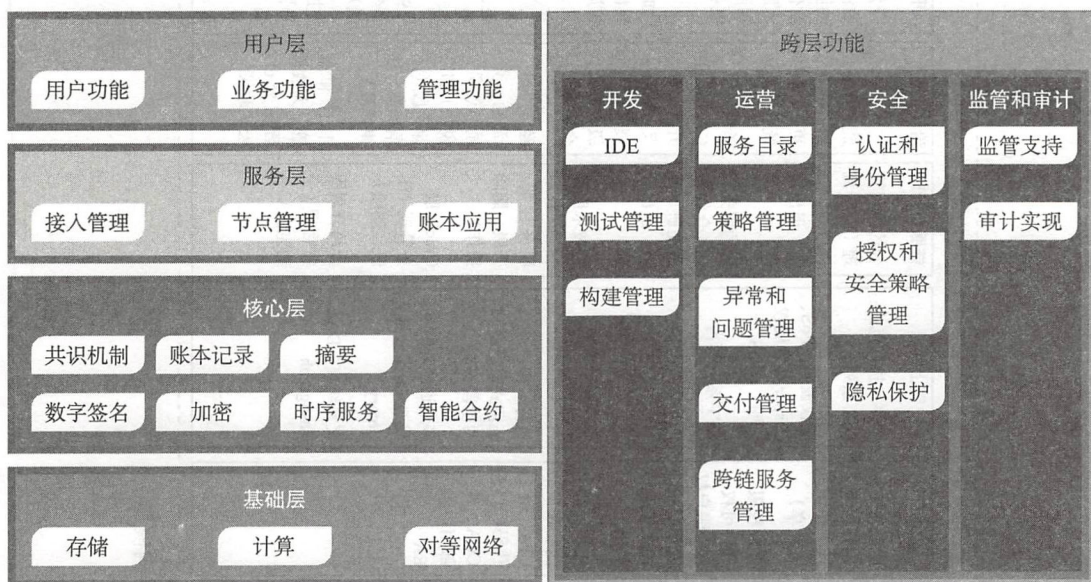


图 3-1 区块链参考架构的功能视图

### 3.3.2 区块链高阶架构模型

为了全景展现区块链生态体系，本节给出了一个面向链合约服务的区块链高阶架构模型，如图 3-2 所示。这个架构模型也体现了未来基于区块链实现高度自动化、智能化、公平守约的虚拟社会生产关系的能力。

面向链合约服务的高阶层次化架构包括三大部分：业务合约和账本、区块链平台服务、UI 界面 /API 接口。

#### 1. 业务合约和账本

在现实世界中，每个人的人生阶段都是处在各式各样的关系契约中，所有人在各种契约约定下参与社会的生产和生活。区块链技术最终要能促进生产关系虚拟化，推动生产力的发展，整个区块链生态系统的核心就是要能支持各种契约，即业务合约，并在相关参与者间共享交易账本。业务合约大到非常复杂的业务合约流程，它要高于企业各自的流程，是各个企业、组织或个人作为流程主体共同参与制定、共同认可的生产关系流程契约；小到“合约服务”，合约服务是在语义层面对业务行为进行抽象的最小契约，合约服务由一组合约动作（action）构成。合约服务可以组合，这同时也组合了不同的业务条款和法律条款或重新定义出合约服务条款。作为抽象的合约服务的具体实现，合约代码可以由不同合约语言编写，合约代码中引用的业务条款和法律条款也都可以由具体的不同实现语言。



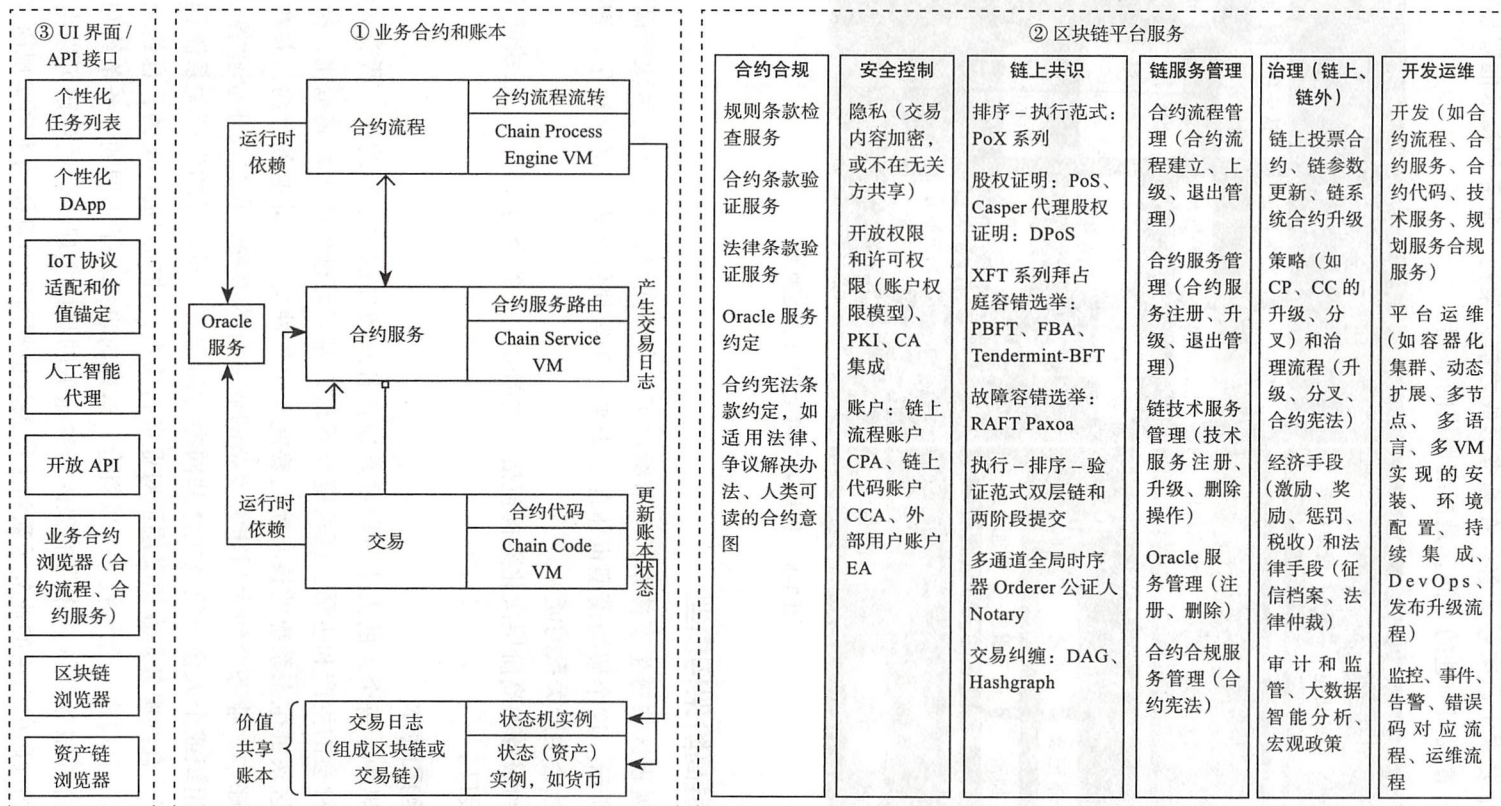


图 3-2 区块链高阶架构模型

### (1) 合约流程

合约流程实现了基于合约服务的一系列固定的、按照既定业务规则和法律条款串联或并联起来的合约动作,通过各个合约动作的完成,实现业务在各个流程参与方的执行,实现由机器流程引擎驱动的价值高速自动创造、自动流转、自动交换。合约流程包括两种类型:一种是全局合约流程,只有一个全局的合约流程和流程状态,所有的参与者都参与其中;另一种是实例合约流程,基于统一的流程模板,可以多个流程实例同时运行,互不干扰。不论哪种流程类型,一旦运行起来就是一个状态机,合约流程在参与方间共享流程状态,也就是共享一致的状态机的状态。业务参与各方在阅读具体的业务合约流程业务规则、法律条款后,签定合约流程契约,合约流程生效。流程可以通过时间触发,可以是链上的事件触发,还可以 Oracle 事件触发。还可以人工触发。初始化流程是一个流程流转交易。如果是人工触发,通过 API 接口或者人机界面输入初始化内容;如果是事件触发,从事件中抽取流程的初始化内容,然后执行流程的实例化。再按照后续节点的类型——人工节点或计算节点,实现通过 UI 界面或 API 的外部输入确认的人工执行驱动,或调用合约服务的具体合约动作实现代码的机器计算驱动,形成业务价值交易。流程引擎调用业务条款检查服务和法律条款检查服务,获得业务节点的业务规则状态和法律条款满足状态,按照既定的流程规则,引擎驱动流程判断流向下一(多)个节点。业务流程在各个节点流转时,由流程引擎驱动形成一个个的流程流转交易,业务流程合约参与方通过共享流程流转交易实现流程状态共享的目的。作为流程验证者的参与者在本地的沙盒中执行对流程流转交易的验证,实现流程状态机一致的状态流转。

### (2) 合约服务

合约服务作为业务合约的主体,定义了业务合约最基本的服务内容。每一个合约服务就是一种最小的完整概念层面的业务语义定义。合约服务定义了完成业务所需的一个或多个合约动作,每个合约动作定义了输入状态和输出状态以及要执行的业务逻辑命令。合约服务本身和每一个合约动作,以及其中用到的任何合约条款、法律条款都需要用人类语言进行清晰明确的描述,必要时提供形式化语义描述避免出现语言理解偏差。每一个合约动作的执行形成一个明确的合约价值交易,实现状态资产在依附的价值主体上可验证的变化——这种状态变化在既定的业务条件和法律条件上是确定的。合约流程引擎在执行流程节点流转时,按照流程定义对合约服务动作的调用,自动进行服务动作的执行,从而产生业务价值交易。这是一种内在的合约服务调用。合约服务的调用也可以是外在的,即由用户通过 UI 界面基于外部 API 接口实现调用。合约服务是自包含的,可以是一种组合服务,即对现有的合约服务动作进行组合,形成新的服务动作业务语义。合约服务的可组合化有利于基于已有的业务合约定义,通过快速构建新型的业务合约进行生产关系创新,实现不同产业价值服务的零距离整合。为了实现合约服务的自包含化,并支持服务流程的编排,以及服务计算容器化、分布式、可扩展的架构部署要求,合约服务需要定义成无状态的。所有的合约动作以及其中的各个命令都是纯业务逻辑描述,或者是对其他公共服务逻辑的



引用。当签约用户或流程调用合约服务时，合约服务会进行服务路由，基于链服务管理的路由规则，选择特定的合约代码实现来具体验证执行业务价值交易。

### （3）合约代码

合约服务的每一个合约动作都会产生业务价值交易。业务价值交易会在合约服务的合约代码实现上进行执行和验证。作为交易验证者，合约服务可以有多种实现，如不同合约语言的实现、不同合约提供商的版本实现、不同利益相关方的合约代码实现等。这有利于所有的合约服务参与者去中心化，并在抽象的业务层面就达成共识。任何不符合服务合约语义的合约实现在测试时就可以被识别，即使在生产运行时个别实现出现不一致的情况，也会被共识算法排除出去，并被区块链平台识别出来。合约代码实现了合约动作定义的输入状态和输出状态和要执行的一组业务逻辑命令计算。命令是最小的执行单元，可以是调用一个技术服务，如生成 zkSNARK 证明、环签名、执行一次 Oracle 服务证明调用，也可以是输入和输出状态检查、逻辑计算、合约规则检查、法律条款服务检查等。经过所有的命令执行，如果输入状态可以确定性地得到输出状态，那么合约服务的发起者就可以形成一个业务价值交易建议，而其他参与者可以对这个业务价值交易建议进行验证。当业务价值交易被打包进区块或组织进交易链后，根据共识算法的确定性规则，交易最终生效。对于合约流程产生的流程流转交易，也是经过相同的方式，由流程执行建议者计算出状态迁移交易建议，由流程执行验证者进行本地流程流转验证，共识通过的交易被打包进区块或交易链。

### （4）价值共享账本

合约流程的流转会产生流程流转交易，合约服务的执行会形成业务价值交易，所有的这些交易日志，按应用、联盟团体组织被分类成区块链或者交易链，形成不可更改和抵赖的数据结构，在各个参与方之间形成一个统一的状态账本。价值共享账本需要以高效、规范的方式进行数据组织，包括交易日志和账本状态，以便于流程状态（状态机）和资产状态数据库的快速更新，也便于对历史交易进行快速查找和回溯。另外，价值共享账本底层还需要有共享的通信机制，如使用各种 P2P 算法，便于相关方进行基于权限的相关交易数据的快速同步。

## 2. 区块链平台服务

区块链平台服务提供所有区块链平台层面的公共服务，平台服务同具体的业务无关，是可以为所有业务合约所共享的服务。各种平台服务可以是链上的，也可以是链外的，一同构成一个区块链平台不可或缺的能力。区块链平台服务主要包括合约合规、安全控制、链上共识、链服务管理、治理（链上、链外）、开发运维。

### （1）合约合规

合约合规服务将那些公共的合规性要求抽取出来，形成各个业务都通用的规则条款检查服务、合约条款验证服务、法律条款验证服务、Oracle 服务约定、合约宪法条款约定。合约宪法指明了合约纠纷适用的规则、争议解决办法，以及人类可读的合约意图等。一个实际的区块链平台能够同现实社会经济并行运行的一个前提条件就是同现实社会保持一致

的合法合规性。由于区块链所建模对象的特殊性,建模的是生产关系和资产价值状态,所有的区块链业务合约,不论是合约流程还是合约服务,都必须服从现实社会法律法规的约束要求。不论是把现实世界中心化的合约去中心化建模成虚拟世界的业务合约,还是基于区块链新型的生产关系新创造的业务合约,合法合规依然是根本。法律和规则都可以以最低粒度的条款内容存在,为了支持业务合约的快速搭建和创新,将通用的合约条款、规则条款、法律条款实现为一种服务,基于对条款服务的引用和基于条款服务的组合,可以实现更高粒度的合规合法性验证要求。业务合约可以基于这些合约规则、合约条款、法律条款和组合的合规合法性验证要求自动进行交易的合规性验证。对于那些无法由代码实现的验证内容和合约意图,可以通过人类可以理解的方式,按照合约宪法指定合约运行所依赖的现实世界法律,指定无法在链上解决的争议在现实世界的解决办法。合约流程和合约服务如果是使用现实世界数据输入的,为了达到确定性运行,所有验证人的验证执行都必须依赖相同的 Oracle 服务或者交易各方都认可的 Oracle 服务。Oracle 服务的约定说明也是各个合约流程或合约服务规格说明的内容。

## (2) 安全控制

区块链在平台安全层面需要设计隐私模型和权限模型。

同现实世界人们需要一定的隐私性和匿名性一样,区块链虚拟世界也需要提供相应的隐私保护给用户,用户才会表达出真实的想法,用户也才会有一定的安全存在感,这样的虚拟生产关系才更符合实际。隐私和监管并不矛盾,在一定程度保证用户隐私和匿名性的情况下,并且让用户知道自身的隐私是真的被保护的情况下,用户才会真正开展符合实际需求业务并配合监管,在需要的情况下公开和上报自己的交易。这一方面有利于虚拟生产关系下的生产繁荣,另一方面也有利于监管的开展。对于公有链,出于网络的安全考虑,往往需要交易无关方对交易内容执行验证,必须采用新的隐私模型,让用户身份信息同用户的交易信息隔离,使用户身份得到保护,甚至采用零知识证明 zkSNARK 算法仅向验证者提供证明。验证者可以在只看到加密交易数据,而不是开放交易内容下完成验证,这样可以做到绝对的交易身份隐匿。然而由于网络的参与方都是受控的,所以许可链防范网络攻击的安全需求没有公有链那么高,共识机制只需确保技术层面的一致和完备,交易业务层面的验证可以只在交易相关方进行验证,这样可以保证交易无关方看不到任何交易内容,即使是加密的交易内容也看不到。

现实世界对权限也有实实在在的需求。对于可编程货币,除了被现实法律剥夺了相关权利的人外,都应该享有操作的权限;对于可编程金融的各种金融业务合约,就需要存在不同金融产品的风险控制模型和权限控制模型;对于可编程社会区块链的各种应用,更需要存在完整的权限控制模型,以控制不同身份主体、不同私钥主体对合约服务的操作权限。公有链是一种开放权限的设计思路,不会显式地设定不同参与人的操作权限,只会采用黑名单机制。而许可链是一种白名单机制,有非常严格的准入机制,只有允许的参与方才可以参与被允许的业务合约。许可链通常采用 PKI 基础设施,通过自身的 CA 机构同企业现有的权限管



理系统进行集成，如 LDAP、AD 服务器，进而控制不同的人具有不同的合约操作权限。

区块链由于采用公私钥机制进行交易，不论采用哪种数据模型，都存在最小粒度的基于私钥的账户概念。区块链存在对合约流程和合约服务两种类型交易（流程流转交易和业务价值交易）的建模数据，所以会存在两种类型的合约相关的账户：合约流程账户（Contract Process Account, CPA）和合约服务账户（Contract Service Account, CSA）。另外，外部用户也会有自己的私钥账户（User Account, UA），外部用户账户会参与到合约流程账户和合约服务账户相关的活动中，而合约流程账户会依赖合约服务账户执行相关的合约动作调用。不同的合约流程和合约服务可以是全局单例，也可以是多实例的，实例化后的相互关系会非常复杂。许可链可以借助现有的权限体系进行控制。而对于公有链，权限模型可以不必采用传统的签名授权机制，我们可以把这三种账户按照使用关系和依赖关系组织成树状结构，采用 Merkle 证明的方式进行权限证明验证。

### （3）链上共识

共识机制是区块链建立信任的基石。不同类型的区块链出于不同的考虑，如网络抗攻击的安全性、参与节点的可控性、共识性能、交易确定性、网络环境等，会选择不同的共识算法，如 PoW、PoS、DPoS、BFT 及其各种变体（如 PBFT、Tendermint）、宕机故障容错（RAFT、Paxos），以及全局时序器（Orderer）、公证人（Notary）机制、两阶段提交（2PC）、三阶段提交（3PC）或者采用各种共识算法的组合，如 PoS+BFT。共识的内容包括账本的规范化（如何组织区块、组织交易链），交易的确定性执行结果，交易的非双花唯一性，交易的顺序完备性（逻辑顺序和时间顺序），以及保证网络安全稳定运行的其他信息（如数据可用性）。另一方面，共识机制的运行又不应同账本的规范化和交易的验证紧密绑定在一起。这也遵从关注点分离的架构原则，有利于区块链平台整体的模块化、插件化、容器化的运行，也有利于增强整个平台的横向可扩展性。共识机制的设计同平台的性能和可扩展性密切相关。在设计共识机制时，除了使用一定的共识算法和共识算法组合，也可以自定义一些协议以提高性能和可扩展性，这是两个重要关注点。增加整个平台的可扩展性是公有链当前面临的重大技术问题。许可链由于存在更加安全可控的运行环境假设，在可扩展性方面可以设计得更加灵活，如交易相关方参与验证交易有效性和完备性、共识算法只做技术性的共识、保证交易唯一性、节点以容器方式无状态运行在可扩展的虚拟基础设施架构上。

### （4）链服务管理

区块链平台服务一个很重要的能力体现就是对于链上服务的高效可靠的管理。所有区块链的业务合约的正常稳定运行都依赖于这些注册的链服务。这种重要性使得链服务的管理需要遵从区块链治理体制和治理流程规则。链服务包括以下几种类型。

#### 1) 合约流程管理。

合约流程管理包括对合约流程的建立、版本升级、退出的管理。这里管理的是合约流程的模板。初始化一个合约流程时，会基于模板建立一个合约流程实例，后续合约流程的

流转都是基于这个合约流程实例运行的。可能存在多个版本的合约流程实例同时运行。用户可以选择签约某个合约流程。签约就代表用户认同合约流程业务规定的法律条款、业务流程和业务规则。一旦某个用户账户绑定在合约流程实例账户上运行,就一直要运行到整个合约流程实例完全结束,可以提供退出子流程中途退出,或者通过治理流程让所有参与者选举主动结束整个合约流程实例。

#### 2) 合约服务管理。

合约服务管理包括对合约服务的注册、版本升级、退出的管理。每一个合约服务都是一个有完整业务意义的、抽象的合约规约,其中每一个合约动作都代表了不同利益方共同遵守的价值约定。一旦签约加入合约服务,就从法律意义上确认了这样的价值约定。

#### 3) 链技术服务管理。

链技术服务管理包括对链技术服务的注册、版本升级、删除操作。不论何种 IT 系统,都一定存在一些公共的技术服务,区块链平台也不例外。合约服务和合约流程运行时所依赖的公共技术服务,如生成链平台相关的 zkSNARK 证明、验证签名,如果发布成链服务的形式,就可以完成基于无状态服务的计算可扩展性,特别适用于计算密集型的技术服务高负载运行时,可最大并行度地支持合约服务的验证执行,也有利于采用特定的硬件加速技术服务。

#### 4) Oracle 服务管理。

Oracle 服务管理包括 Oracle 服务的注册、版本升级、删除操作。Oracle 服务是虚拟世界同现实世界的桥梁,很多业务合约的运行都离不开来自于现实世界的信息,所以必须提供统一的 Oracle 服务,供所有的验证人进行运行时验证,才能保证交易验证的确定性,否则共识无法形成,区块链信任机制失效。信任是需要传递的,各个验证人信任区块链机制,也必须信任 Oracle 服务,所以 Oracle 服务必须要具有公信力。在现实社会中,一个组织的公信力(如政府)来源于完善的监督罢免机制(如两院议会)。Oracle 服务横跨两个世界,所以必须在两个世界都要设立对它的监督制约机制。虚拟世界的监督机制设计,如存入大额抵押金成为 Oracle 服务提供方,成立赏金猎人的监督机制,一旦被赏金猎人发现非法行为,如提供同实际情况不符的 Oracle 证明,即被没收所有抵押金,吊销 Oracle 服务资质,记录征信档案,在现实世界也需要做出相应的惩罚。

### (5) 链上治理和链外治理

任何不同利益主体参与的活动,从长期稳定发展的角度来看,都需要配套的治理策略和机制保障。区块链作为多利益主体参与的动态变化系统,其架构处于不断演进过程中,运行的业务合约也不断发展,还会有利益驱使的恶意行为,以及有组织的黑客攻击行为,运行的业务合约和交易也都存在监管和审计的需求。为了让区块链可以稳定安全地运行,特别是对于公有链,需要从公平正义的基本法理出发,预先设计出完整的博弈经济模型和社会化治理机制。博弈经济模型可以保证区块链的参与者以不同的角色,积极高效自觉地参与和维护区块链的生产、管理和治理,对符合区块链整体利益的行为进行激励,对正义



行为进行奖励，对恶意行为进行惩罚，使用经济手段阻断黑客攻击，让攻击行为得不偿失。另外，要模仿现实世界对经济活动征收税收，用于整个区块链平台的治理。为了能高效公平地推进区块链的治理，可以预先设计出扩展性良好的底层治理机制，如设计底层的链上投票合约。基于这个底层机制可以进行相关平台重大事项的社会化投票公决，如区块链主宪法的更改、链参数的更新、链系统合约的升级、业务合约（合约流程、合约服务、合约代码）以及合约法律的升级。

对于那些无法通过链上解决的治理问题，或者需要现实世界配合解决的问题，以及那些还无法预见的问题，需要设定链外治理的策略和机制，如对于确认的业务合约中的恶意行为或黑客行为，除了经济手段惩罚，还可以上报征信系统，诉诸现实世界的法律手段。

一个稳定运行的区块链系统就会形成一个经济和金融体系，离不开对其中运行的经济交易的持续审计和监管，以杜绝违法合约和交易行为，如反洗钱交易。每一个业务合约的接入方作为主体，如果为其他人提供代理接入的，需要负责对客户进行尽职调查，做到KYC监管要求。区块链的有效治理还离不开基于区块链交易数据的大数据智能分析。由于区块链是一个经济系统，可能还需要基于分析结果施以适应经济规律的宏观政策。

#### （6）开发运维

一个成功的区块链平台就是一个多利益主体参与的生态系统，每一个参与主体（政府、企业、组织、个人）都有可能参与到平台的开发和运维工作中来。

设计和开发人员可以参与到基础平台层服务的设计开发，也可以实现业务合约的规格制定和开发。这其中会涉及架构人员、业务人员、法律人员、技术人员、监管人员等各种专业人员。一个业务合约的设计和开发，首先需要由业务人员、法律人员和架构师完成完整的业务合约规约的制定。不同价值主体可以共同完成或由一方完成后讨论，形成合约共识，制定出完整的合约流程和合约服务规格说明书。然后，再由不同的参与主体自行开发实现或委托实现，可以不断迭代提炼出通用的服务，如法律条款检查服务、通用技术服务、各方在自行开发实现时充分利用平台已有的成熟的通用服务以提高实现效率和服务稳定性。参与各方可以采用不同的语言实现合约服务逻辑，以保证合约服务语义层面的一致性和合约的分布性。规格设计人员和开发人员可以设计和开发合约流程、合约服务、合约代码、技术服务、规则服务和合规服务。

每一个参与主体，特别是验证节点都可以参与到区块链的平台运维中来。运维行为包括对运行节点服务的容器化集群，提供动态扩展能力，安装多语言多VM（虚拟机）实现节点，支持多节点并行运行、并行验证。生产运维需要有完善的流程，如生产环境配置流程和发布升级流程。面对区块链日新月异的变化，可以利用DevOps进行持续开发、持续集成的新开发运维体制和自动化测试部署流程。

对生产系统需要能够进行监控，进行事件记录，对重要事件发出告警；对于告警错误码，需要预先制定处理流程；针对区块链系统和业务，还需要预先制定出正常情况和异常情况下的运维流程。

### 3. UI 界面和 API 接口

这部分是提供整个区块链服务对外的交互接口，包括提供给人的 UI 界面和提供给其他信息系统或人工智能代理的 API 接口。交互的内容主要包括以下几个方面。

#### (1) 个性化任务列表

价值主体加入某个合约流程后，如果合约流程的某个业务流程节点需要主体的输入和确认，就会转化成对这个主体的界面交互请求。用户需要在一个业务界面中输入必需的内容，或者确认系统提供的业务信息，并使用主体的业务操作私钥进行签名，以表明主体的身份，让业务合约得以继续进行。主体可以同时加入多个合约流程，这就会存在一个任务列表，需要主体逐个进行界面操作完成。

#### (2) 个性化 DApp

每一个业务合约都可能是一个 App，多个业务合约一起也可以是一个 App。用户、用户的 IoT 智能终端，或者用户的人工智能代理、加入的每一个业务合约（合约流程或合约服务）都是一个业务应用，所以需要为用户提供定制化的分布式 App，以满足用户的个性化需求。这需要建立一个大一统的 App 基础平台，在其上提供各式插件式的个性化小应用，就如同微信上搭载的各式各样的小程序。虽然表面展现方式相同，其实内核完全不同。这是个 DApp，里面运行的小应用也完全是分布式的，用户自己的身份信息自己管理，没有集中的机构可以完整获得。所有小应用的交易和授权都是基于用户各个应用的私钥进行的，只由用户本人控制。

#### (3) IoT 协议适配和价值锚定

区块链一个大的应用方向就是同物联网的结合。物联网的各种终端要实现智能化自动制造、智能化自主服务，就需要将它们绑定到虚拟世界，传统的 IoT 中心化控制架构是无法直接反映社会化生产和服务要求的。区块链作为一个虚拟的经济社会，维持了虚拟的经济生产关系，让 IoT 智能终端参与到区块链群体中，参与到具体的区块链合约流程和合约服务中，由社会化的区块链机器自动驱动 IoT 终端进行自动化的生产和服务，并引入人工智能代理加速人工处理，可以极大地提高生产力。区块链需要同 IoT 的协议进行适配，以确保双向交易的无障碍流通。另外，为了在虚拟世界建模现实世界的价值生产、转移和交换，将现实世界真正融入虚拟世界的生产关系合约中，需要为现实世界生产的产品和服务价值在虚拟社会分配一个价值锚定标签。如同虚拟世界拥有了私钥就可以锁定价值一样，在现实世界，也需要有一套可行的方案将虚拟世界的价值锚定标签，植入现实世界的产品和服务中去。不同的产品和服务可能需要不同的锚定机制。通过价值锚定标签，现实世界价值的生产、转移和交换就可以无缝融合进虚拟世界的生产关系合约流程与服务中去了。有了完善的价值锚定机制，可以真正实现价值在虚拟世界和现实世界的无障碍流通与融合。

#### (4) 人工智能代理

价值主体可以使用人工智能代理帮助其完成合约流程的自动流转和合约服务动作的自动发起，虚拟世界高速运转的生产关系需要这样的角色。随着人工智能的发展，人工智能代理



也能够胜任基本的基于规则和用户习惯的操作。另外,结合大数据智能分析,在设定一定的业务目标后,可以由人工智能代理主动发起一些优化的交易,减少人工操作成本,提高整个合约服务的运行效率,可以预见性地优化资源配置,减少整个社会化生产的资源浪费。

#### (5) 开放 API

整个区块链平台对于可以开放的或者可以权限开放的接口,都提供标准的 API,允许外部系统或人工智能代理进行访问和操作。区块链的各种业务合约(合约流程、合约服务)信息,区块链的各种交易结果,当前流程状态,资产状态,或者区块链的交易发生证明,资产存在证明,链上治理接口,也都可以用 API 的方式向外部系统提供。通过 API 接口,也可以进行各种业务合约的操作,如人工处理的提交、合约动作交易的提交等。

#### (6) 业务合约浏览器

通过业务合约浏览器,用户可以看到权限范围内所有可参与的业务合约,包括合约具体的规格化内容,如合约流程、合约服务各动作、合约具体规则、合约法律条款、合约宪法、治理规则等。

#### (7) 区块链浏览器

区块链浏览器可以浏览所有的区块以及权限许可的交易内容,可以对可浏览的交易进行回溯查看,可以从不同的维度进行交易、流程和价值资产的审查。

#### (8) 资产浏览器

资产浏览器允许用户以统一的视角看待用户关联的所有合约资产。资产浏览器可以同个性化 DApp 整合在一起,让用户可以看到当前各个参与合约流程的当前状态,各个合约服务的状态资产,以统一的视图帮助用户进行交易的优化决策。

### 3.3.3 区块链跨链本质与架构模型

把整个现实社会都搬到一个区块链上是不现实的,现实社会本身也是分产业分经济领域进行价值创造,然后通过市场实现不同产业和不同经济领域的价值交换。每一个独立区块链维护了自己独立的价值经济体系,跨链区块链是连接独立区块链的中枢,承载了不同价值体系区块链价值交换的功能。商品要实现交互需要有价格,价格来源于商品自身的价值,取决于供求关系,而供求关系是靠市场搭建的,所以,为了实现不同区块链“商品”的价值交换,在跨链区块链上会出现各种价值交易市场。跨链区块链上每一个价值交易市场就是一个跨链合约服务。价值不会凭空产生也不会凭空消失,跨链设计也必须遵从人类自古以来的经济规律。跨链的本质是价值等价交换,任何违背这个基本原则的设计最终都会失败。

图 3-3 中独立区块链的架构模型已经在“区块链的架构模型”一节中讲过了。所有独立区块链如果需要支持跨链价值转移或交换,就需要存在外链合约服务。外链合约服务同普通的合约服务没有本质的区别,也是一种合约服务规约,不同之处在于合约的制定者会提供一组公开声明的跨链交易公钥地址。需要进行跨链交易的主体可以把自己拥有的一定数量的价值体转移到外链合约服务指定的公钥地址上,并指定跨链交易内容,如希望交换另一个区块链上一定数量的价值体,并把交换后的价值体转到自己在另一个区块链上的公钥地址上。

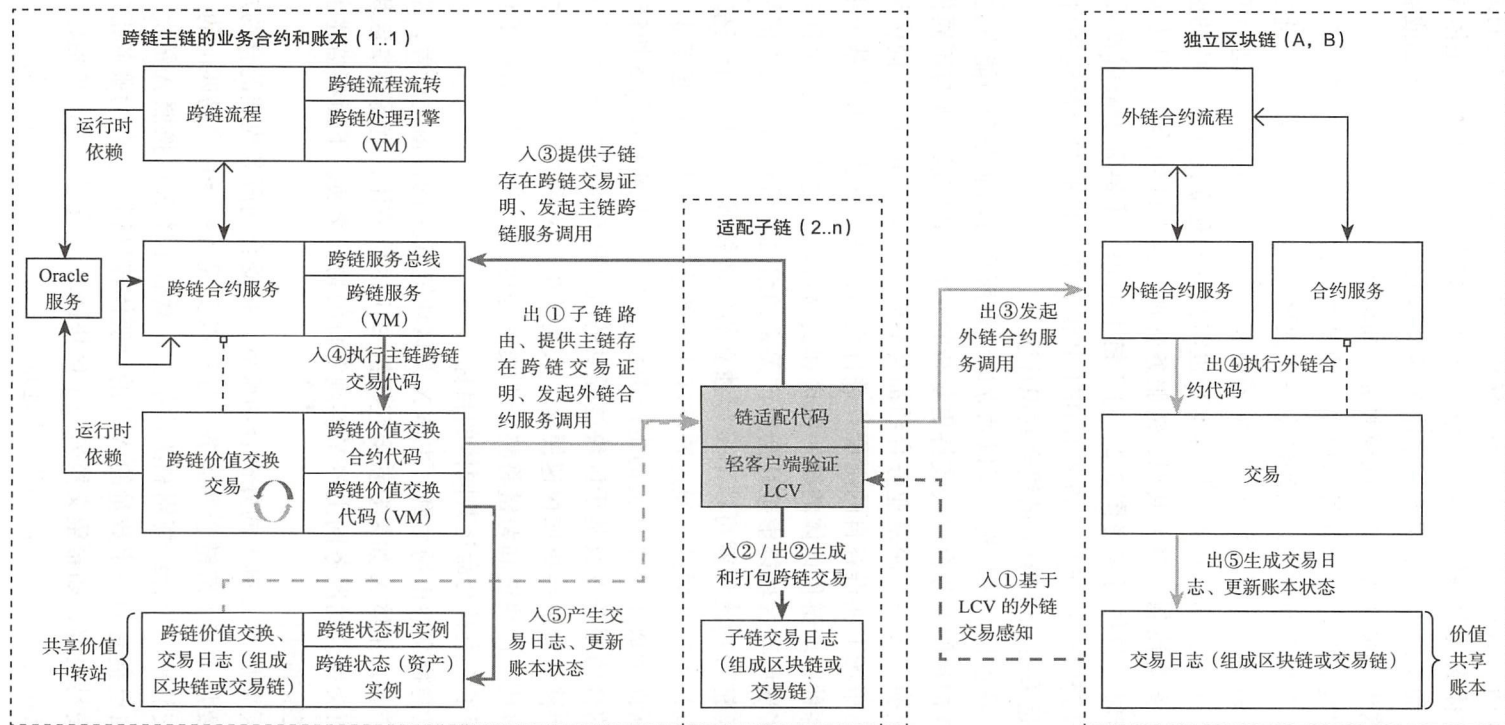


图 3-3 区块链跨链架构和跨链过程



这里假定存在两个独立区块链 A 和 B，存在一个主体 X 和主体 Y，它们都拥有两个链上的私钥地址。主体 X 是区块链 A 上的价值生产者，如农民生产粮食，主体 Y 是区块链 B 上的价值生产者，如工厂生产工业品。主体 X 希望购买区块链 B 上的产品或服务，如工业品，主体 Y 希望购买区块链 A 上的产品或服务，如粮食。

跨链区块链主要有两种类型的链组成。一种是主链，跨链主链只有一个；另一种是适配子链，适配子链至少存在 2 个，由跨链主链连接各个适配子链，各个子链之间没有信任关系，而是通过主链进行信任的传递。适配子链和主链按照设定的协议进行交互，以达到信任传递和交易传递的目的。跨链区块链本身也需要有同独立区块链一样的区块链平台服务，如合约合规、安全控制、链上共识、链服务管理、链上链外治理、开发运维，这些在图 3-3 中都做了省略，具体可参考图 3-2。对于链上共识，主链和子链需要采用比 PoW 更加高效的算法实现跨链交易交互，如采用 BFT 共识算法。目前两个跨链平台（Cosmos 和 Polkadot）设计都是采用 PoS+BFT 的混合共识算法。

跨链区块链本身也是个区块链，所以独立区块链所具有的业务合约能力它也应该具有，但基于跨链区块链构建的业务合约会支持更复杂的业务，其可以实现与不同价值区块链的连接，进行价值交换。每一个跨链业务合约都会形成一个交易市场，不同区块链的不同价值体系在这个交易市场上获得各自的定价并进行交易，极有可能会形成基于主链代币或者主权加密通货的各种区块链价值体的统一报价和交易市场。更高级的跨链区块链可以通过跨链合约流程，实现所有区块链虚拟社会生产关系的组合。假设每个独立区块链是一个独立的经济领域，跨链合约流程就可以串接起独立的经济领域，成为一个完整的产业链条。跨链区块链本身也是可以互联的。通过跨链区块链的连接，可以串接起工业、农业、服务业等各行各业，从而构成了整个社会的生产关系。生产生活都关联到区块链虚拟社会上，基于区块链提供的合约服务以及基于区块链提供的机器驱动业务流程，结合 IoT 和人工智能，价值生产、转移和流通会更加快速便捷，人类的生产关系也会更加优化协调，生产力可以得到彻底的解放。区块链和跨链将整个人类对等地关联在一起，去除了任何的信息不对称性，真正实现了人类社会命运共同体的目标。

我们这里仅以物物交换市场为例。主体 X 是区块链 A 上的价值生产者，主体 Y 是区块链 B 上的价值生产者，主体 X 如果要获得区块链 B 上的价值体，就需要拿区块链 A 上的价值体通过跨链价值交换合约服务同主体 Y 实现等价的物物交换。下面将描述主体 X 和主体 Y 实现价值等价交换的过程。

首先主体 X 需要加入 A 链上的外链合约服务，接受合约服务规定的合约规则和法律条款。主体 X 还需要加入某个跨链合约服务，如可以实现  $A \longleftrightarrow B$  交易匹配的一个跨链合约服务，接收跨链交易市场的合约规则和法律条款。然后主体 X 需按照 A 链上外链合约服务的合约规则，把自己拥有的一定数量的 A 链的价值体转移到外链合约服务指定的公钥地址上，并指定跨链交易内容，如希望交换另一个区块链 B 上设定数量的价值体，并把交换后的价值体转到自己在另一个区块链上的公钥地址，从而开始整个跨链交易。

后续的交易过程如下。

(1) 入①基于 LCV 的外链交易感知

适配子链的轻客户端验证 (LCV) 会不断同步区块链 A 的区块头, 其对于区块链 A 上的外链合约服务公开的公钥地址敏感, 一旦发现存在公钥地址的交易, 就认为存在跨链交易请求。

(2) 入②生成和打包跨链交易

由链适配代码将区块链 A 上主体 X 指定的跨链交易请求内容 (用链 A 上一定数量的价值体兑换链 B 上一定数量的价值体到指定公钥地址上) 生成一个子链交易, 并且打包进子链区块。

(3) 入③提供子链存在跨链交易证明, 发起主链跨链服务调用。

链适配代码基于 Merkle 树给出一个跨链交易请求在子链上的存在性证明, 并按照跨链协议, 封装出发往主链的跨链服务调用。

(4) 入④执行主链跨链交易代码

主链的跨链服务总线, 验证交易在子链上的存在性证明, 分析主体 X 的跨链交易请求内容, 将跨链服务调用路由给具体的跨链价值交换合约。同样过程, 主体 Y 的跨链交易请求 (用链 B 上一定数量的价值体兑换链 A 上一定数量的价值体到指定公钥地址上) 也被发往相同的跨链价值交换合约。

(5) 入⑤产生交易日志, 更新账本状态

跨链价值交换合约的代码实现, 会进行所有的  $A \longleftrightarrow B$  交易匹配, 形成一个 A 链价值体同 B 链价值体的买卖市场深度, 一旦可以匹配上主体 X 和主体 Y 的交易请求, 就形成一个匹配交易, 用以封装 A 链和 B 链价值体在主体 X 和 Y 之间达成交换的结果。跨链价值交换合约本质上就是一个场内交易所。

(6) 出①子链路由, 提供主链存在跨链交易证明, 向适配子链发起外链合约服务调用

跨链价值交换合约实现代码, 会提供一个交易主体 X 和 Y 的跨链匹配交易在主链上的存在性证明, 分别向链 A 和链 B 的适配子链发送转账指令交易, 一个指示往 A 链 Y 主体指定的公钥地址转移一定数量的价值体, 一个指示往 B 链的 X 主体指定的公钥地址转移一定数量的价值体。

(7) 出②生成和打包跨链交易

这两个适配子链分别将各自的转账指令交易记录日志, 并打包进各自的子链区块。

(8) 出③发起外链合约服务调用

链适配代码向各自对应的独立区块链上的外链合约服务发起转账指令交易。A 链的适配子链会向 A 链的外链合约服务发送一个转账交易, 指示从合约的公开地址往 Y 主体指定的公钥地址转移一定数量的价值体。B 链的适配子链也会向 B 链的外链合约服务发送一个转账交易, 指示从合约的公开地址往 X 主体指定的公钥地址转移一定数量的价值体。

(9) 出④执行外链合约代码

A 链的外链合约服务会执行合约代码, 生成交易, 把由合约控制的、转账指令要求的一



定数量的价值体转移给 Y 主体指定的公钥地址。B 链的外链合约服务会执行合约代码，生成交易，把由合约控制的、转账指令要求的一定数量的价值体转移给 X 主体指定的公钥地址。

#### (10) 出⑤生成交易日志，更新账本状态

一旦交易被打包进区块，按照链的交易确认特性，最终主体 X 获得了 B 链的价值体控制权，主体 Y 获得了 A 链的价值体控制权。

跨链区块链也会提供用户 UI 界面和 API 接口，用户所有在跨链区块链合约服务上执行的交易都可以通过跨链用户界面和 API 接口获得当前的执行状态，即查看用户在交易所挂单状态和交易市场的买卖深度，甚至可以让用户基于私钥按照市场供求关系重新挂单。跨链区块链可以提供基于独立区块链上的外链合约服务的抵押机制，在对应的适配子链上换取相同数量的抵押区块链的价值体 token 或筹码，业务主体拿抵押的子链上的价值体 token 参与主链的业务合约流程。这种跨链的生产关系，基于各个主体抵押的各自区块链上的价值体（也可以是现实世界价值锚定）配置生产资料，开展合约生产，最后分配生产产品价值。如果跨链区块链有自己内生的代币，也可以基于交易市场（合约）完成到内生代币的价值兑换，主体拿着跨链代币加入跨链合约流程或跨链合约服务的虚拟生产关系进行生产和价值交换。

### 3.3.4 区块链部署模型

区块链部署架构模型如图 3-4 所示。从区块链实现虚拟化、自动化和社会化协作生产的目标出发，基于关注点分离的架构原则和层次化的架构模式给出的区块链架构模型，从设计时就需要考虑平台的可用性。以当前的计算架构，采用多台大型主机的银行服务或者采用分布式架构的互联网服务才能支撑起整个社会范围的交易并发，这还只是若干银行、几大互联网公司共同提供的集中式交易。区块链共识就意味着冗余计算，区块链又是建立在密码学上的计算，本身就需要耗费大量的计算能力，要能够提供满足目前银行和互联网服务性能的区块链虚拟计算，就需要目前所有银行主机和分布式服务计算能力的若干倍才可以。如果要实现连接现实社会的自动化流程驱动的生产，整个社会的计算能力还需要有极大的提高。区块链架构要想实现在整个社会范围内的实用化就必须实现功能模块的松耦合，需要能够支持分布式并行计算、支持密码学专用硬件加速，甚至支持连接高性能计算中心的第三方计算。

目前区块链架构模型设计成验证服务和平台共识服务分离，业务验证服务的合约流程和合约服务以及实现代码分层服务化解耦，业务合约服务同公用的合规合法检查服务，技术服务以服务化的方式解耦，区块链交易日志、状态的规范化逻辑同平台共识服务逻辑分离解耦。所有这些功能逻辑的服务化、无状态化，目的就是确保服务的横向分布式部署扩展能力，实现服务容器化按需动态扩展，充分利用当今云计算的发展成果。另外，按照参与业务主体紧密程度、业务相关性、业务性能要求、隐私要求的不同，形成多个子链、侧链的高度分离以提高整个区块链的并行处理能力。

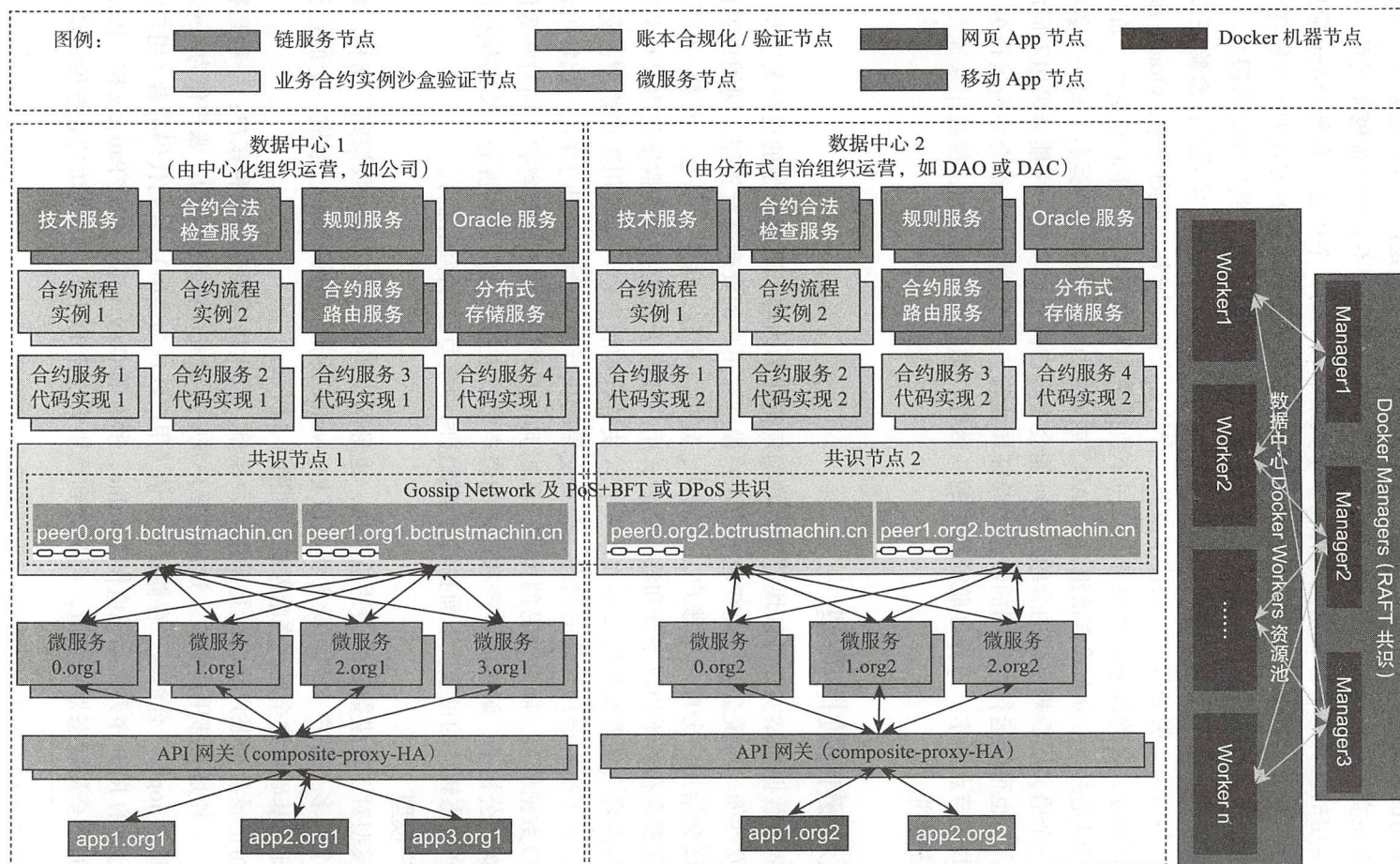


图 3-4 区块链部署架构模型



真正实用的区块链共识节点上要运行大量的应用，需要满足巨大吞吐量要求，并且响应时间也需在实用可接受的范围，共识节点所需要的计算能力不是个体能够承受的，所以未来一个实用的区块链平台一定是运行在多个数据中心上的，个体通过各种 DApp 应用参与链上合约业务。数据中心会提供大量容器资源，以动态可扩展的方式为区块链各个功能模块提供服务运行所需的计算资源和存储资源，从前端的 DApp，到后端的微服务，再到区块链共识服务、账本服务、各种业务合约（合约流程、合约服务）实现的沙盒验证节点，以及各种公共的链上服务节点，如技术服务、合约合法检查服务、规则服务、Oracle 服务、分布式存储服务、合约服务路由服务等。一个数据中心可能是由一个中心化组织（如公司）运营，也可以是由一个分布式自治组织（DAO<sup>⊖</sup>或 DAC）依据自治合约运行。每一个数据中心对于同一个语义层面规格化的合约服务可能会有自己的代码实现，可能会采用不同的合约编程语言，也可能运行在不同的沙盒中验证和执行。每个数据中心都会并行运行多个账本副本和共识节点副本，以保证验证结果的一致性，提高系统可用性，提高出块速度，避免遭受经济惩罚。

### 3.4 区块链的数据模型

区块链的数据模型实际是区块链平台数据规范化的内容。数据模型的设计关乎整个区块链的并发性能。传统金融行业普遍采用账户模型对用户的资产进行管理。同用户签订的每一份协议会给用户分配一个账户，如活期、定期、贷款、理财、基金、证券、保险等，每一份协议都存在若干协议条款和法律条款，用户或金融机构按照条款发起协议相关的交易，账户的内容会随着交易的发生而改变。区块链合约状态也可以延用账户模型：每一个合约账户记录了所有合约管理的状态，合约交易的执行驱动合约状态的变化。比特币首先引入 UTXO 数据模型，随着交易的执行，私钥拥有对资产状态进行解锁，同时将等值资产锁定到新的公钥上，解锁的资产和锁定的资产可以进行等值汇总和拆分，交易执行的同时也是资产状态转移的过程。下面我们分别就这两种模型进行详细说明。

#### 1. 账户模型

账户模型对于建模数据的表达最直接，也形同于传统的“应用 + 数据库”结构。账户模型将一个或多个账户实例的所有状态以 key-value 的形式组织起来，形成一定的存储结构，并以此为基础构建全局状态哈希树，如图 3-5 所示。

以太坊记录各个账户状态就是采用账户模型。以太坊需要建模两种账户：一种是智能合约账户，一种是外部用户账户。这两种账户统一存储成一种数据结构，账户作为 key，账户余额、账户 nonce、合约账户存储根哈希值（用户账户为空串）、合约代码哈希（用户账户为空串）组成的 RLP 序列化编码值作为 value，所有的 (key, value) 构建 Patricia 树。Patricia 树根就是账本全局状态根哈希值，会参与到区块头哈希的计算。以太坊的数据模块请参考

⊖ [https://en.wikipedia.org/wiki/Decentralized\\_autonomous\\_organization](https://en.wikipedia.org/wiki/Decentralized_autonomous_organization).

以太坊相关章节的描述。

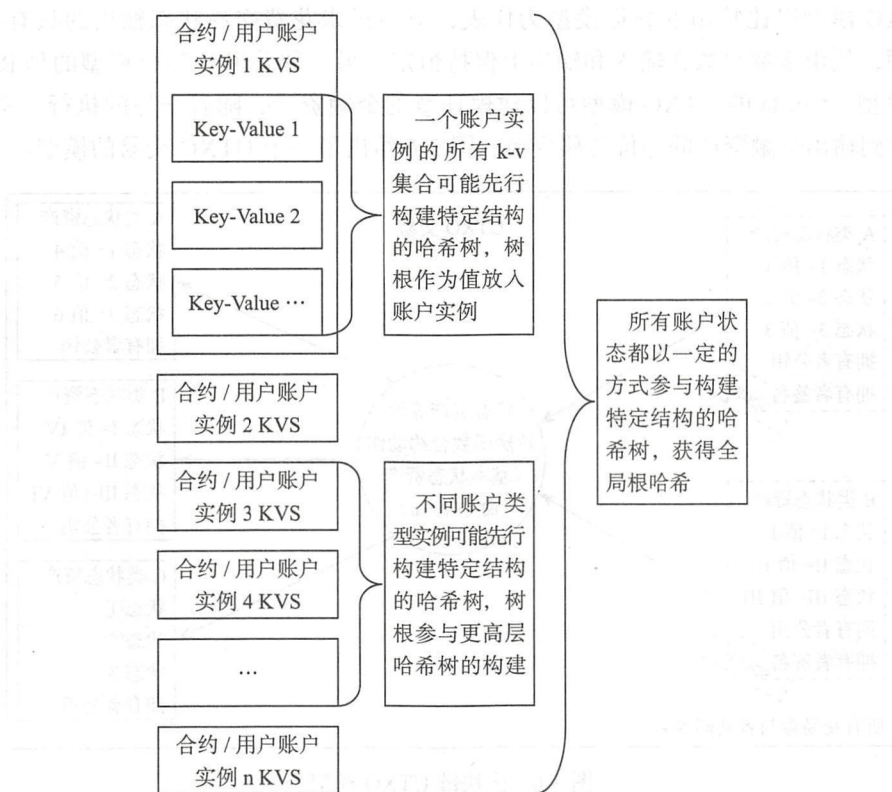


图 3-5 区块链账户模型

Fabric 需要记录各个 chaincode 状态。每个 chaincodeid 都以 key-value 的形式组织键值存储 KVS。这里一个特别的地方在于 key 值具有版本的概念，一个交易会依赖于一个特定的读 key 值版本集合，形成新的写 key 值版本集合，基于状态值集合的版本控制实现了对多个状态值写交易的唯一性和完备性控制。这样对于两组互不相关的状态的写操作就可以实现交易的并行执行，而不会像以太坊由统一账户 nonce 控制，对同一个账户的写交易必须串行执行。Fabric 把若干个 chaincodeid 的 KVS 组成一个存储 bucket（桶）分区，再由若干个 bucket（桶）编成一组，每组的 bucket 进一步进行哈希（计算），由哈希值组成新的 bucket，直到形成世界状态（World State）根哈希。根哈希参与区块头哈希的运算。

合约账户实例可以是合约流程账户实例或合约服务账户实例。所有的账户实例状态一同构建特定结构的哈希树，这样所有区块链的共识状态都参与到全局状态树根哈希的运算中来，而全局状态树的根哈希又会参与到当前区块头的生成。通过验证节点对区块所包含交易的执行，修改验证节点本地的全局状态树中各个账户实例相关的各个状态值，重新计算出全局状态树的根哈希，同区块头中的根哈希做比对，就从交易层面和全局状态树层面进行了闭环业务印证。



## 2. UTXO 模型

UTXO 模型以比特币的数据模型为代表,只不过未花费交易状态输出的只有一个值:代币金额,代币金额总数在输入和输出上保持恒定。另一种采用 UTXO 模型的是 R3 Corda 的数据模型。Corda 的 UTXO 模型可以建模任意的金融资产,随着交易的执行,实现输入金融资产到输出金融资产的等价转移变换。图 3-6 给出了一个 UTXO 交易的模型。

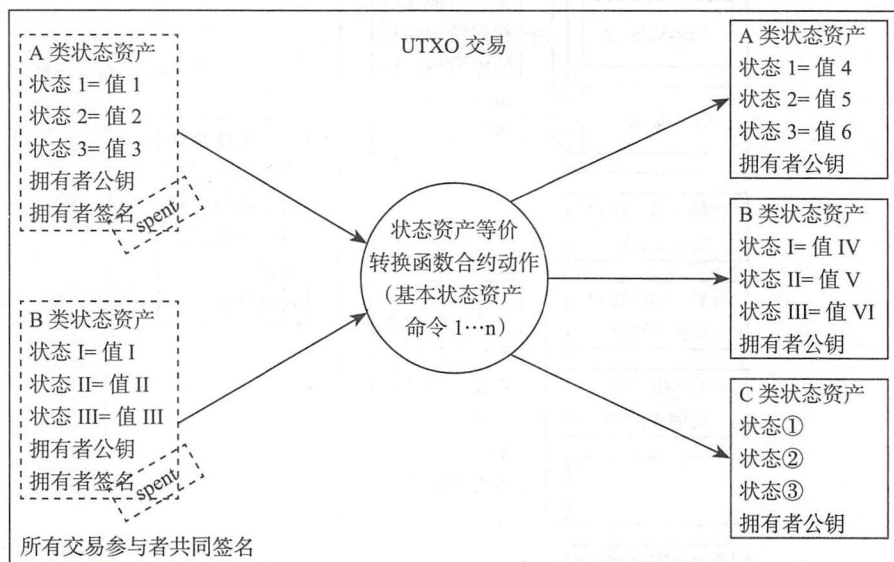


图 3-6 区块链 UTXO 模型

UTXO 交易会把带有拥有者签名的若干输入状态资产消费掉,由状态资产等价转换函数,也就是合约动作执行一系列的基本状态资产命令,执行新的状态资产的输出,新的状态资产输出给交易输入要求的公钥地址上。每一个基本状态资产命令负责某一类状态资产转移所需的检查,按照合约动作的条款约定执行该类状态资产的销毁和新状态资产(包括新的状态值)的生成。交易的执行过程可能需要依赖于 Oracle 服务,如获得资产转换汇率、现实世界条件等。

UTXO 模型的优势:

- ❑ 不可变账本静态数据快照易于分析;
- ❑ 交易并行化处理能力(交易主体只需存在足够的未花数据项);
- ❑ 交易时序不那么重要了(无须像以太坊全局维护每个账户的 nonce);
- ❑ 冲突解决仅归结为双花问题,对共识算法要求降低到对一组条件是否满足的判断。

UTXO 模型的不足:

- ❑ 不可变状态如果是数字类型,表现形式不自然,需要通过拼接未花来满足支付交易需求;
- ❑ 对于钱包交易管理者需要有余额和结算的概念,这些都需要基于 UTXO 数据模型运算得出,而且需要较为复杂的代码逻辑处理;

□ 在严格要求处理时序的场景下，由于交易的并行化处理而变得艰难。

UTXO 最大的特色在于交易的并行处理能力：属于同一个转出账户的所有交易可以并行执行，只要同一个交易不出现在同一个区块当中。UTXO 模型的交易本身就代表状态，无须构建全局状态树，只需要构建交易哈希树就可以完成状态验证。区块链当前的状态可通过构建未花交易输出对象的 UTXO 数据库获得。另外一个特点是 UTXO 数据库不会随着时间而膨胀，新的交易未花输出同时伴随着旧的交易未花输出的消费，所以基本上可以保持相对稳定的状态，这不同于账户模型随着网络交易参与账户的增多，账户数据库会不断膨大。

### 3. 混合模型

账户模型和 UTXO 模型可能会被结合起来应用，以充分利用两种模型的优势。

以太坊的扩容方案——分区（Sharding）就是在账户模型的基础上使用 UTXO 模型进行跨分片的通信。以太坊在分片方案中，如果按照账户地址的前 2 个字节进行地址分片，0x0000 开头的地址交易在一个分区中进行处理，0x0001 开头的地址交易在另一个分区中处理。依此类推，总共可以有  $2^{16}$  个分区，分区节点只进行自己分区交易和状态的处理，从而可以提高整体的交易吞吐量。而分区同分区之间的通信则采用类似 UTXO 的方式进行处理，如图 3-7 所示。

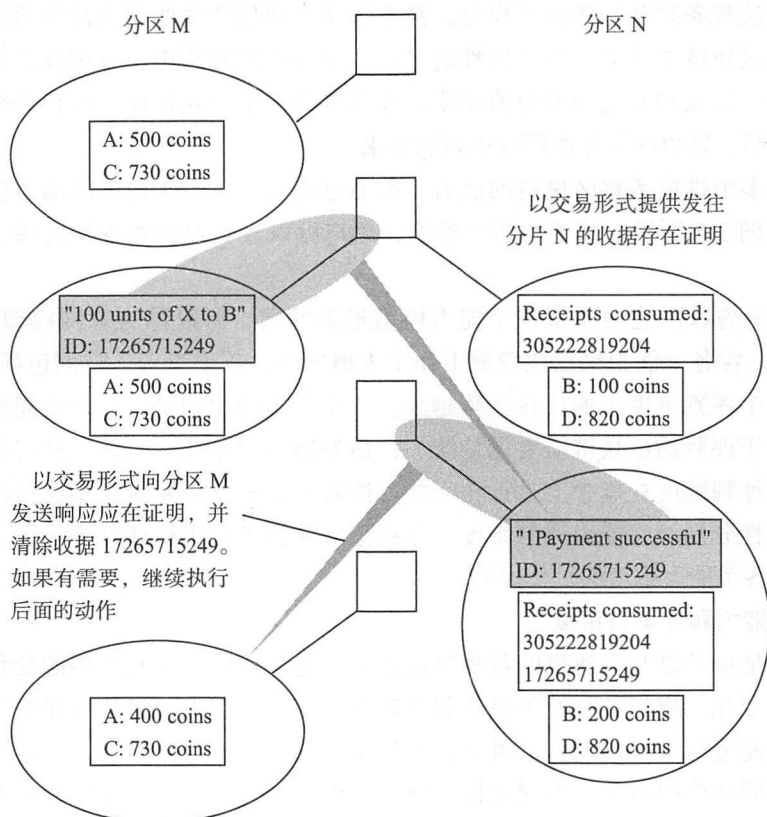


图 3-7 以太坊账户模型引入类 UTXO 收据进行分区扩容



利用类 UTXO 收据进行分区扩容的思路简单分析如下。

1) 在分区 M 上发送一个交易, 把 A 账户的余额扣减 100 个代币给分区 N 上的 B 账户, 并生成一个转账给 B 账户的转账收据。收据并不存在于状态上, 但可以为收据提供 Merkle 存在证明, 证明收据确实产生了。

2) 等待前面交易的最终确认, 确保交易确定性地存在于链上。

3) 向分区 N 发送一个交易, 交易包括了上述收据的 Merkle 存在性证明。交易会检查分区 N 中该收据的“未花”状态, 如果收据确实“未花”, 为 B 账户的余额增加 100 个代币, 并在状态中保存该收据“已花”状态。

4) 可选的是, 上一步骤执行结果也可以产生一个收据, 表明该转账给 B 的收据已经成功执行, 然后通过交易发往分区 M, 以允许其基于转账结果进行后续操作。

### 3.5 区块链的能力模型

通过前面章节的叙述, 读者可以了解到, 从功能上来说, 区块链不仅仅是一个技术平台, 一个业务(流程)平台, 更是一个虚拟社会生产关系的运行平台, 涉及社会的方方面面——从法律法规条款到经济博弈模型, 甚至涉及不同信任主体间的政治治理。而从非功能性上来看, 区块链要成为一个实用性的平台, 支持可编程货币、可编程金融、可编程社会的各种应用, 以及相互之间的价值交换, 那么区块链平台在性能、可扩展性、隐私和匿名性、权限控制、易用性等方面都有很高的要求。

为了能从多个维度考察区块链的能力, 本书也给出了一个区块链的能力模型——蛛网能力模型, 如图 3-8 所示。基于此能力模型, 用户可以直观地看到各种区块链在不同能力维度上的差别。

图 3-8 所示的蛛网能力模型这个能力模型形似蛛网, 所以称为蛛网模型。对比特币、以太坊、Fabric 在各个能力维度比较域上作了大概区分, 仅供参考。蛛网包括的范围越大, 就代表其在各个评判维度上的比较优势越大。一个基本的原则是, 各个功能性、非功能性的评判标准对于评判的区块链需要保持统一, 最好能够定量化。另外, 选定各个评判维度时, 需要确保评判维度在概念上是正相交的。读者可以根据需要对模型进行改进, 如对功能性和非功能性分别设立能力评判维度、分别建立蛛网模型进行评判等。以下是对这个区块链蛛网模型各个能力维度的具体说明。

#### (1) 建模能力和业务自由度

建模能力是指可以共识建模的各种对象能力, 包括建模金额和金额的变化、建模各种状态和状态的变化、建模各种业务流程和流转变化的能力。建模能力还包括对建模对象的共识操作能力, 如只能进行金额的调整, 可以进行复杂的业务逻辑处理, 可以进行复杂的业务流程化处理。不同的建模对象、不同建模对象的操控能力使得不同的区块链具有不同的业务共识自由度。



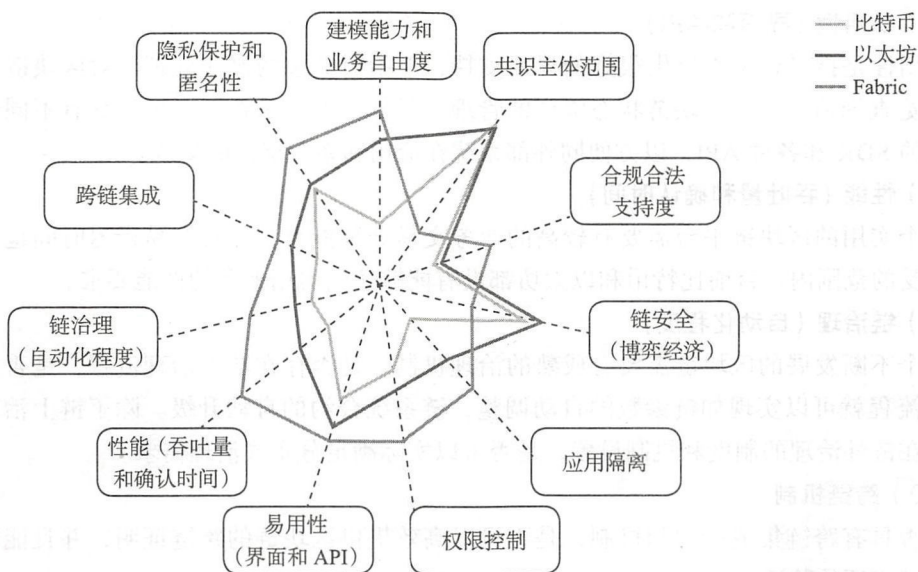


图 3-8 区块链蛛网能力模型

## （2）共识主体范围

共识主体可以是开放主体，即不限制任何人使用区块链平台。共识主体也可以是许可范围内的主体，不同的许可主体参与不同的业务处理。共识主体范围分布越广泛，共识达成的价值也越高。

## （3）合规合法支持度

合规合法支持度是指区块链是否可以反映现实世界的法律法规，是否能够建模现实世界的法律法规，并且用于业务合约的验证执行。一个合规合法支持度越好的区块链，其应用范围也会越广泛。

## （4）链安全（博弈经济）

区块链的安全性取决于很多方面，如交易非双花唯一性、交易完备性、交易不可抵赖性等。这些特性都可以称为区块链的基本要求。然而设计是否具有博弈经济性也是区块链能够健康运行的促进因素。一个能同共识算法很好地配合起来，并且能够规则化地在链上运行的经济策略，可进一步保障区块链的安全性。

## （5）应用隔离

不同类型的应用产生的交易，考虑是不是放在一起运行和验证？是不是打包在一起？对于应用的执行和应用的交易有没有做到一定的隔离？通过隔离，可以实现毫不相干的交易的并行执行和验证，从而解决以上问题。

## （6）权限控制

不同的应用是否可以实现不同的权限控制，或者全部开放权限？一个具有权限控制的应用和系统设计更能符合实际业务要求。





### (7) 易用性 (界面和 API)

易用性是指平台是否提供很好的界面支持, 包括对业务的操作处理、对区块链区块和交易浏览查询追溯、对区块链状态资产的管理。另外, 平台是否可以提供具有不同编程语言支持的 SDK 和各种 API, 以方便同外部系统 in 应用和系统层面的集成。

### (8) 性能 (吞吐量和确认时间)

一个实用的区块链平台需要要有较高的业务交易处理能力, 并且交易延迟时间也是在用户可接受的范围内。目前比特币和以太坊都没有能够达到实用平台的性能要求。

### (9) 链治理 (自动化程度)

一个不断发展的区块链需要要有成熟的治理机制。是否存在链上治理流程, 是否通过链上治理流程就可以实现如链参数的自动调整, 链系统合约的自动升级。除了链上治理, 是否还存在链外治理的制度和机制设置, 是否可以实际满足分布式治理需求。

### (10) 跨链机制

是否具有跨链集成能力和机制, 是否可以高效提供区块链的跨链证明, 并且能够在对方链得到方便的验证。

### (11) 隐私保护和匿名性

是否支持隐私保护, 支持交易内容的匿名性, 是否需要在无关方公开共享交易数据内容, 是否可以提供绝对的隐私保护和匿名性。

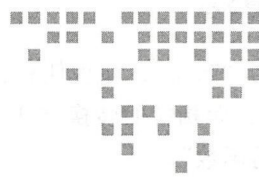
## 3.6 小结

本章从各个维度讲解了区块链的架构, 并且分别就各个典型的区块链架构分析, 包括比特币、Zcash、以太坊、Corda、Fabric、EOS、Cosmos 和 Polkadot。通过大量的区块链实例的架构介绍, 读者可以形成一个整体的对区块链架构的认知。本章还给出了跨链的架构模型, 以及基于实用性、高扩展性的设计目标给出了区块链的部署模型, 描述了区块链的数据模型和能力模型。通过各个维度的架构讲解, 从具体实例到架构抽象, 读者可以全方位地掌握区块链的核心架构, 也对区块链的设计要点有了明确的认识。

### 习题

- [1] 区块链有哪些共性? 又存在哪些差异点?
- [2] 请说明区块链的本质是什么? 跨链的本质是什么?
- [3] 为什么区块链要采用服务化、容器化的架构设计?
- [4] 为什么要为区块链设计博弈经济模型?
- [5] 为什么区块链的治理非常重要?
- [6] 请给出几个区块链的能力模型维度。





## 第 4 章

## Chapter 4

## 密 码 学

作者：唐 屹

密码技术是区块链系统安全性保障的基础技术。在区块链系统中，使用了哈希函数、公钥密码技术等现代密码学技术。

在区块链平台中，保证分布式账本信息的完整性是最重要的。完整性是信息安全的三个基本要素（Confidentiality, Integrity, Availability, CIA）之一。区块链系统的完整性是指系统能够保护账本数据免受未授权的修改，具有检测区块是否被篡改的能力。哈希函数是实现完整性保护的主要工具。区块链系统中至少包含 2 个层级的完整性保护。首先，一组账本数据的全局状态由 Merkle 哈希树所保护，其根哈希存储于区块中。这组数据内的任意信息改变都可能导致一个新的根哈希值，从而导致整个区块哈希值的改变。其次，通过哈希指针的使用，区块的历史，即账本的历史记录，能够得到保护，这使得区块一旦添加到区块链中，其内容就不可改变。

共识机制是区块链系统中维持数据一致性的基础，比特币这类公有链系统应用了 PoW 机制实现全网区块链数据的一致性和安全性。基于哈希函数的工作量证明算法利用了哈希算法的谜题难解特性，每个网络节点利用自身计算资源寻找特定前缀的哈希值以竞争区块记账权，只有完成一定计算工作量并提供证明的节点才可以生成区块。

区块链系统中还采用了公钥密码技术和数字签名技术。不同于传统的对称密码算法，公钥密码算法采用了两个相关的密钥：一个密钥是公开的（公钥），另一个密钥是保密的（私钥）。公钥密码算法的提出，有效地解决了密钥分发传输的问题，而私钥加密 - 公钥解密的应用模式，也带来了有效的数字签名算法。区块链系统中使用了椭圆曲线公钥密码技术以及基于椭圆曲线的数字签名技术，同时使用数字证书完成公钥的分发。





## 4.1 哈希算法

哈希也称为“哈希”，实现任意长度的输入转换成固定长度的输出，这个输出即称为输入的“哈希值”，而相应的转换方法称为“哈希算法”。在不引起混淆的情况下，哈希算法也常称为“哈希函数”。

碰撞是与哈希函数相关的重要概念，体现着哈希函数的安全性。所谓碰撞是指两个不同的消息在同一个哈希函数作用下，具有相同的哈希值。实际上，哈希函数的输入长度是任意长的，而输出长度是固定的，这意味着碰撞是不可避免的。哈希函数的安全性是指在现有的计算资源（包括时间、空间、资金等）下，找到一个碰撞是不可行的。

密码学哈希函数可以视为一类满足安全性需求的哈希函数。它能够在有限合理的时间内，将任意长度的消息压缩为固定长度的二进制串，然而却很难在有限合理的时间内找到这个二进制串的碰撞。以密码学哈希函数为基础构造的哈希算法，在现代密码学中扮演着重要的角色，常用于实现数据完整性和实体认证，同时也构成多种密码体制和协议的安全保障。

### 4.1.1 密码学哈希函数简介

在形式上，哈希函数  $H: X \rightarrow Y$ ，其中， $X$  为定义域， $Y$  为值域，且  $|X| > |Y|$ ，这意味着  $H$  为多对一映射，把任意有限长的输入映射到固定长的输出。哈希函数存在碰撞是必然的。从计算意义上，若把值  $x$  及其哈希值  $H(x)$  唯一地联系在一起，我们希望能够找到这样的哈希函数  $H$ ，满足在计算上寻找碰撞是困难的，这里的“困难”是指最快的算法也需要消耗与枚举差不多的时间和资源才能够找到两个不同的元素具有相同的哈希函数值。

密码学哈希函数  $H$  应满足如下要求。

- 1) 压缩:  $x$  任意长,  $H(x)$  固定长。
- 2) 计算出  $H(x)$ 。
- 3) 像攻击 (Preimage Resistant): 已知  $y \in Y$ , 要找出  $x \in X$ , 使得  $H(x)=y$  是困难的; 也称单向性 (One-way)。
- 4) 二原像攻击 (Second Preimage Resistant): 已知  $x \in X$ , 找出另一个  $x' \in X$ , 使得  $H(x')=H(x)$  是困难的; 也称弱抗碰撞性 (Weak Collision-Resistant)。
- 5) 碰撞性 (Collision-Resistant): 找出任意两个不同的  $x, x' \in X$ , 使得  $H(x)=H(x')$  是困难的; 也称强抗碰撞性 (Strong Collision-Resistant)。

几乎所有针对哈希函数的破解攻击，主要指的都是破坏上面的抗碰撞性。

在计算机工程领域，除留余数法是哈希函数的常见构造方法，其定义为： $H(x)=x \bmod p$ ，其中  $p$  通常为一个素数，显然对于任意的  $x$ ，定义  $x'=x+p$ ，均有  $H(x)=H(x')$ 。因此，简单采用除留余数法构造的哈希函数并非密码学哈希函数。

值得注意的是，找到两个碰撞的消息在计算上是困难的，并不意味着不存在两个碰撞



的消息。例如，若哈希函数的输出值长度固定为 256 位，顺序取  $2^{256}+1$  个输入值为 1, 2, ...,  $2^{256}+1$ ，逐一计算其哈希值，肯定能够找到两个输入值，使得它们的哈希值相同。

一般情况下，依据生日悖论，对于哈希值长度为 256 位的哈希函数，如果随机挑选其中的  $2^{130}+1$  个输入，则有 99.8% 的概率可以发现至少 1 对可产生碰撞的输入值。然而，这样的计算可能非常耗时以至于计算不可行。对于哈希值长度为 256 位的密码哈希函数，要找到碰撞对，平均需要完成  $2^{128}$  次哈希计算，如果计算机每秒能够进行 1 万次哈希计算，则需要约 1027 年才能完成这  $2^{128}$  次哈希计算。

在密码学上还有一个概念是“理论破解”，指的是提出一个算法，使得可以用低于理论值的枚举次数找到碰撞。一般的，依据生日悖论，对于  $n$  比特长的哈希函数，找到冲突的计算复杂度为  $2^{n/2}$ 。MD5 曾是广泛使用的一种密码学哈希函数。中国学者王小云教授等人在 2004 年将 128 比特的 MD5 哈希函数的碰撞算法复杂度从  $2^{64}$  降到了  $2^{42}$ ，这个研究成果表明了 MD5 算法的理论破解。为应对这类理论破解，必须及时添加应用限制条件，或者重新选用更为安全的哈希函数标准，以保证密码学哈希函数的安全应用。

在不引起混淆的情况下，我们此后所指的哈希函数即为密码学哈希函数。

#### 4.1.2 哈希函数的性质及应用

哈希函数最重要的性质就是抗碰撞。利用这一性质引申出来的信息隐藏和难题友好的性质，被充分利用到了区块链系统的构造中。

##### 1. 抗碰撞

如前所述，碰撞是与哈希函数相关的重要概念，两个不同的消息在同一个哈希函数作用下，若产生相同的哈希值，即形成碰撞，哈希函数  $H$  的抗碰撞性使得寻找两个不同的、能够产生碰撞的消息在计算上是不可行的。

哈希函数的抗碰撞特性常被用来进行完整性验证。由于抗碰撞性，我们可以把哈希值作为原始输入消息的指纹（因为很难找到另一个消息经哈希运算之后得到相同的哈希值）。如果原始输入消息在传输过程中被篡改，那么经过哈希函数处理后得到的新哈希值就会和原来的哈希值不一样，这样很容易就能发现消息的完整性在传输过程中受到破坏。

对区块链来说，哈希函数的抗碰撞性可以用来做区块和交易的完整性验证。在区块链中，第  $n+1$  号区块的头部信息中会存储着第  $n$  号区块信息的哈希值。如果能拿到第  $n$  号区块的信息，任何用户都可以通过简单的比对计算得出第  $n$  号区块的哈希值，并与存储在第  $n+1$  号区块中的第  $n$  号区块哈希值进行比较，检测第  $n$  号区块的信息完整性。

##### 2. 信息隐藏

原像不可逆是单向函数的重要特征。通俗地说，原像不可逆是指知道输入值可以很容易通过函数计算出函数值，但知道函数值却难以计算出原来的输入值。

哈希函数的原像在计算上是不可逆的，这意味着依据哈希函数的输出是不能计算出该





哈希函数的输入的, 即已知  $H(m)$ , 试图计算出原始消息  $m$  在计算上是不可行的。因此, 从计算意义上来看, 哈希函数隐藏了原始消息  $m$ 。

为描述哈希函数的信息隐藏特性, 我们先引入最小熵的概念。最小熵 (min-entropy) 是信息理论中衡量某个结果的一个可预测性指标, 用于度量最坏情形下的随机性。而通常意义下的信息熵则是度量平均情形的随机性。当  $x$  从某集合中随机均匀选择时, 两者没有区别。但是当  $x$  不是均匀选择时 (或者说  $x$  有一部分信息被泄露), 区别就出现了, 存在一些分布信息熵高、但容易预测的例子。当一个  $x$  具有  $k$  比特的最小熵时, 该  $x$  被猜中的可能性不大于  $2^{-k}$ 。高最小熵指的是变量呈均匀分布 (随机分布)。

因此, 若对消息  $m$  进行哈希计算时引入一个随机前缀  $r$  ( $r$  来自高最小熵的概率分布数据集), 则给定哈希值  $H(r||m)$ , 试图恢复出消息  $m$  在计算上是不可行的, 这意味着该哈希值隐藏了消息  $m$ 。

承诺方案 (Commitment Scheme) 被认为是密码学领域中一类重要的密码学基本组件, 承诺具有隐藏性和绑定性。

承诺模型可以看作一个数字化的密封信件。如果 Alice 想承诺某个信息  $m$ , 则她可将  $m$  放入一个密封的信封内, 形成隐藏了  $m$  的数字信件 (承诺的隐藏性); 而若她想公开这个信息, 则只需打开信封, 公布  $m$ 。而一旦信封打开, 任何人都能验证 Alice 所公布的  $m$  是否就是她最初承诺的信息  $m$  (承诺的绑定性)。

一般而言, 承诺方案包含 2 个算法。

1) 承诺值计算  $\text{commit}(m, r)$ : 输入消息  $m$  和随机值  $r$ , 返回承诺值  $c(=\text{commit}(m, r))$ 。

2) 承诺验证  $\text{verify}(c, m, r)$ : 输入承诺  $c$ 、消息  $m$  和随机值  $r$ , 若  $c==\text{commit}(m, r)$ , 返回“真”, 否则返回“假”。

显然, 哈希函数可以用于构造承诺方案。例如, 若定义  $\text{commit}(m, r) = H(r || m)$ , 利用哈希函数  $H$  的抗碰撞性和信息隐藏性, 承诺的隐藏性和绑定性均能成立, 可以实现承诺方案。

### 3. 谜题难解

在形式上, 哈希函数  $H$  称为谜题难解 (Puzzle Friendliness)。若  $k$  选自一个高最小熵分布, 对于每个  $n$  比特的哈希值  $y$ , 不可能以小于  $2^n$  的时间复杂度找到一个  $x$ , 使  $H(k || x) = y$ 。这说明如果想产生一些特殊的哈希值  $y$ , 同时哈希函数的一部分输入固定、另一部分输入随机, 则很难找到那样的随机值, 使得计算出来的哈希值正好等于  $y$ 。

考虑一个由哈希函数构成的谜题: 已知哈希函数  $H$ , 一个高最小熵分布的值  $v$  以及目标范围  $T$ , 寻找  $x$ , 使得  $H(v||x) \in T$ 。

求解上述问题等价于需要找到一个输入值, 使得输出值落在目标范围  $T$  内。这里可以定义  $T$  为哈希函数值域的一个子集。例如, 如果哈希函数  $H$  的输出为  $n$  比特, 那么输出值可以是任何一个  $0 \sim 2^n - 1$  范围内的值, 可以定义  $T$  为  $0 \sim 2^k (k < n)$  范围内的值。目标范围  $T$  的大小决定了解这个谜题的求解难度。如果  $T$  包含所有  $n$  比特长的串, 那么求解等价于计



算一次哈希值；但如果  $T$  只包含一个元素，则这个求解是最难的，相当于给定一个哈希值，找出其中的一个原像。事实上，由于  $v$  具有高最小熵分布，这确保了除了随机尝试  $x$  值以完成  $0 \sim 2^k$  范围内的搜寻外，并没有其他更好的求解方法。

哈希函数的谜题难解性是基于 PoW 的共识算法的设计基础。一个简单的谜题设计是寻找给定前缀的哈希值，例如，寻找给定的字符串  $v$  为 blockchain，后接整数值串  $x$ ，希望连接后的字符串的 SHA-256 哈希值以  $d$  个十六进制 0 开头的，即  $\text{SHA-256}(\text{"blockchain"}||x) < \text{SHR}(2^{256}-1, d*4)$ ，其中  $\text{SHR}(a, n)$  表示对无符号数  $a$  右移  $n$  位。按照这个规则，由  $x=1$  出发，递增  $x$  的值，我们需要经过 2 688 次 SHA256 计算才能找到前 3 个十六进制位均为 0 的哈希值，而要找到前 6 位均为 0 的 SHA-256 值，则需进行 620 969 次 SHA-256 计算。这样找到的符合特定要求的哈希值的原像，可以作为共识算法中的工作量证明。

### 4.1.3 区块链中的哈希函数

在哈希函数的应用历史上，MD5 和 SHA-1 是两个广泛使用的哈希函数，这也使得两者成为密码学者研究的重点。如前所述，攻击哈希函数的基本思路是设计优于生日攻击复杂度的算法，以快速找到哈希函数的碰撞。MD5 早在 2004 年就被确认容易找到碰撞。2008 年，Marc Stevens 等人通过所设计的选择前缀哈希碰撞工具，创建了伪造的 SSL 证书，显示了使用 MD5 哈希函数签发 SSL 证书的危害性。而 2012 年火焰（Flame）蠕虫就利用了 MD5 的碰撞伪装了微软签发的证书。

SHA 是一类由美国国家标准与技术研究院 (NIST) 发布的密码学哈希函数。1993 年，SHA 家族第一个成员 SHA-0 发布；两年后，SHA-1 发布；之后，另外的 4 种变体：SHA-224、SHA-256、SHA-384 以及 SHA-512 相继发布。这些算法也被称为 SHA-2。

2017 年，Google 表示其已成功破解了 SHA-1 算法，放出了一个概念验证，两份特制 PDF 文档拥有相同的 SHA-1 哈希值，但内容上却不尽相同。正是由于 MD5 和 SHA-1 存在的这些安全性问题，这两个哈希函数已从安全性要求高的应用场合消失。

SHA-3 算法在 2015 年 8 月正式发布。值得注意的是，SHA-3 并不是要取代 SHA-2，因为 SHA-2 并没有出现明显的弱点。出于对 MD5、SHA-0 和 SHA-1 出现成功攻击的忧虑，NIST 感觉需要一个与之前算法不同的、可替换的哈希算法，NSA 于 2007 年正式宣布在全球范围内征集新一代（SHA-3）算法设计。2012 年，在进入第三轮评选的五个候选算法 Keccak、BLAKE、GrøSTL、JH 和 SKEIN 中，Keccak 算法最终获胜并成为官方唯一的标准 SHA-3 算法。

比特币系统中使用了两个密码学哈希函数，一个是 SHA-256，另一个是 RIPEMD160。RIPEMD160 主要用于生成比特币地址。SHA-256 是构造比特币区块链所用的密码学哈希函数，同时比特币地址的生成过程中也用到了 SHA-256。无论是区块的头部信息还是交易数据，都使用了这个哈希函数去计算相关数据的哈希值，以保证数据的完整性。同时，在比特币系统中，基于寻找给定前缀的 SHA-256 哈希值，设计了 PoW 的共识机制。然





而, SHA-256 算法被认为容易导致硬件矿机和矿池的出现, 引发算力集中, 违背去中心化原则。

以太坊使用 Ethash 算法实现工作量证明机制。以太坊工作量证明基于改进的 Dagger-Hashimoto 算法, 这类算法难以通过设计特殊的 ASIC 芯片实现。由于以太坊的开发时间恰逢 SHA-3 标准的发展时间, SHA-3 在最终的哈希算法的填充上做了更改, 因此以太坊的 sha3\_256 和 sha3\_512 哈希不是标准的 SHA-3 算法, 而是一个变体, 通常称为 Keccak-256 和 Keccak-512。

SCRIPT 算法被用来设计比特币 (Bitcoin) 的工作量证明机制。SCRIPT 算法最早被用于设计基于口令的密钥生成函数, 在设计时就考虑了抵御基于硬件的密钥搜索的攻击模式, 这使得算法需占用更多的内存, 花费更长的计算时间, 并使并行计算变得异常困难。与 SHA-256 相比, SCRIPT 算法具有更强的抵御矿机的能力。基于 SCRIPT 算法的比特币的成功, 催生了各种各样的算法创新, 使得每一个使用创新算法的币种出现都能引起一阵波澜。

考虑到现存的大量可用哈希函数, 人们不满足于使用单一的哈希函数。2013 年, 夸克币 (Quark) 首先使用了 6 种公认的安全哈希算法 (BLAKE、BMW、GROESTL、JH、KECCAK 和 SKEIN) 进行了 9 轮的哈希运算设计工作量证明机制。达世币 (DASH, 前身是暗黑币 Darkcoin) 则设计 X11 算法使用了 11 种哈希算法 (BLAKE、BMW、GROESTL、JH、Keccak、SKEIN、LUFFA、CUBEHASH、SHAVITE、SIMD、ECHO)。

Equihash 是一种内存依赖型的工作量证明算法, 其理论依据是广义生日悖论问题, 其算力大小主要取决于拥有多少内存。该算法能够避免少数拥有专用采矿设备的矿工对挖矿过程的中心化的趋势。2016 年 4 月, 密码货币 Zcash 采用了这一算法。Zcash 官方认为, 该算法在很短时间很难出现矿机, 同时由于广义生日悖论是一个已经被广泛研究的问题, 很难有人或者机构能够对算法偷偷进行优化。Equihash 算法非常容易验证, 这有助于在手机上设备上实现 Zcash 轻客户端。

CryptoNote 为比特币以及门罗币采用的应用层协议, 采用基于哈希的工作量证明算法 CryptoNight。这个算法适用于普通 PC 的 CPU。由于目前并没有可用的专用挖矿设备, CryptoNight 只能通过 CPU 挖矿。

## 4.2 Merkle 树

Merkle 树 (默克尔树, 也称哈希树) 是一类基于哈希指针的树。利用 Merkle 树可以实现信息快速方便的完整性验证。

### 4.2.1 哈希指针

哈希指针 (见图 4-1) 是一种数据结构, 它除了指示信息存储的位置之外, 带有这些信



息的哈希函数值存储在一起。使得哈希指针不仅可以检索信息，也可以检测所指向信息的完整性。

区块链可以看作一类使用哈希指针的链表，如图 4-2 所示。这个链表链接一系列的区块，每个区块包含数据以及指向表中前一个区块的指针。在区块链中，前一个区块指针由哈希指针所替换，因此每个区块不仅告诉前一个区块的位置，还提供一个哈希值去验证这个区块所包含的数据是否发生改变。

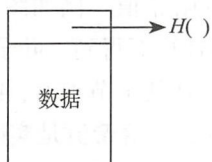


图 4-1 哈希指针



图 4-2 哈希指针链

代码清单 4-1 描述了一个简单的区块定义。

代码清单 4-1 简单的区块定义

```
typedef struct block {
    uint256      hPrevBlock;
    datatype     data;
    struct block *ptPrevBlock;
} block_t;
```

指针 ptPrevBlock 指向前一个区块，data 则表示区块中所记录的数据信息，而 hPrevBlock 表示前一个区块中除 ptPrevBlock 域外的长度为 256 比特的哈希值。一般来说，若 p 为指向当前区块的指针，q (= p → ptPrevBlock) 为指向 p 的前一个区块的指针，则  $p \rightarrow hPrevBlock = H(q \rightarrow hPrevBlock \parallel q \rightarrow data)$ 。

我们可以使用上述的定义去构造一个基于区块链的防篡改数据系统。在这个系统中，区块链中的各个区块被用来存储数据，区块通过哈希指针链接，新区块追加在链表的尾部。同时，定义一个初始区块，其内容不可改变。希望这个数据链表中的某个区块的数据被篡改，则系统能够检测出来。

记区块 k 除 ptPrevBlock 域外的数据为 blk[k].hPrevBlock 和 blk[k].data，依据区块链的定义，有  $blk[k+1].hPrevBlock = H(blk[k].hPrevBlock \parallel blk[k].data)$ 。假定具有强大计算能力的攻击者篡改了区块 k 的数据 blk[k].data，这将导致哈希函数  $H(blk[k].hPrevBlock \parallel blk[k].data)$  产生新的值。由于后继区块 k+1 中的 blk[k+1].hPrevBlock 存储了区块 k 的哈希值，简单地对比就能发现新计算出的值与区块 k+1 所存储的哈希值不一致，篡改就能被发现。注意到，为掩盖这种不一致性，攻击者可能在篡改 blk[k].data 的同时也篡改了 blk[k].hPrevBlock，以保证新计算出来的  $H(blk[k].hPrevBlock \parallel blk[k].data)$  的值与 blk[k+1].hPrevBlock 存储的值一致。问题是 blk[k].hPrevBlock 的改变，区块 k-1 涉及的 blk[k-1] 内容也需要改变，这意味着攻击者至少需要连续改变前 k-1 个区块中的 hPrevBlock 值来保持



一致性。然而，这种改变在回溯到初始区块时，由于初始区块的数据内容不可改变，使得数据对比最终出现不一致，这样攻击者篡改数据的行为就可发现。因此，只需单个哈希指针，就能保证整个链表存储内容的完整性，从而达到防篡改的目的。

#### 4.2.2 Merkle 哈希树

哈希树是一类基于哈希指针的二叉树或多叉树，其叶子节点上的值通常为数据块的哈希值，而非叶子节点上的值是该节点的所有子节点的组合结果的哈希值。例如图 4-3 为一个 Merkle 哈希二叉树，节点 A 的值必须是通过节点 C、D 上的值计算而得到。叶子节点 C、D 分别存储数据块 001 和 002 的哈希值，而非叶子节点 A 存储的是其子节点 C、D 的组合的哈希值。这类非叶子节点的哈希被称为路径哈希值，而叶子节点的哈希值是实际数据的哈希值。

该 Merkle 树的根节点 ROOT 的值则由其子节点 A、B 的内容组合的哈希值所决定。

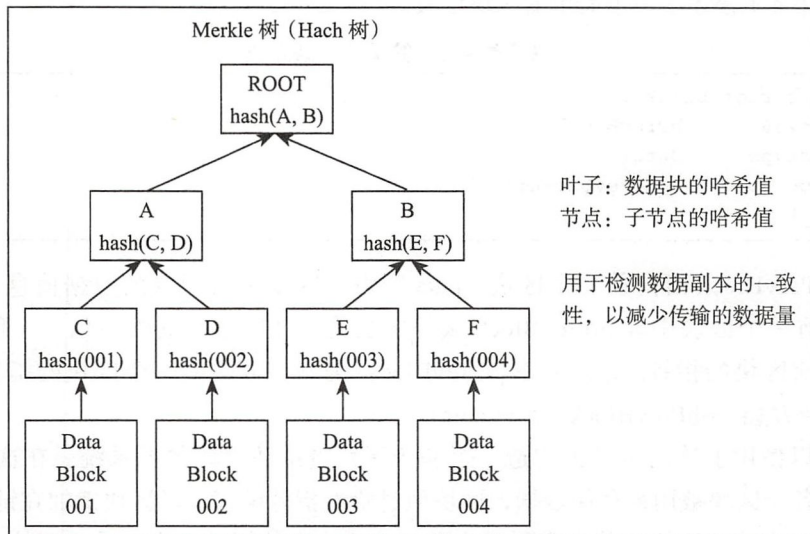


图 4-3 Merkle 哈希二叉树

在处理完整性验证的应用场景中，特别是在分布式环境下进行这样的验证时，Merkle 树会大大减少数据的传输量以及计算的复杂度。以图 4-3 为例，若 C、D、E 和 F 存储了一组数据块的哈希值，当把这些数据从 Alice 传输到 Bob 后，为验证传输到 Bob 上的数据完整性，只需要验证 Alice 和 Bob 上所构造的 Merkle 树的根节点值是否一致即可。如果一致，表示数据在传输过程中没有发生改变。假如在传输过程中，E 对应的数据被人篡改，通过 Merkle 树很容易定位找到（因为此时，根节点、B、E 所对应的哈希值都发生了变化），定位的时间复杂度为  $O(\log(n))$ 。

利用一个节点出发到达 Merkle 树的根所经过的路径上存储的哈希值，可以构造一个 Merkle 证明，验证范围可以是单个哈希值这样的少量数据，也可以是验证可能扩至无限规

模的大量数据。

代码清单 4-1 所定义的区块存在一个问题，即计算区块的哈希值时使用了区块所记录的数据 data。注意到在实际应用中，这样的 data 大小规模可能存在很大的差别。例如，如果 data 对应的是某个时间间隔内的日志，则日志记录数可能相差很大，为统一计算时间开销，可以考虑引入 Merkle 哈希树，以 Merkle 哈希树的根值替代 data 值。代码清单 4-2 即定义了这样一个引入 Merkle 根哈希的区块。

代码清单 4-2 带 Merkle 哈希值的区块定义

```
typedef struct mblock {
    /* 区块头部 */
    uint256      hPrevBlock;
    uint256      hMerkleRoot;
    /* 区块记录数据 */
    datatype     data;
    struct mblock *ptPrevBlock;
} mblock_t;
```

### 4.3 公钥密码算法

公钥密码体制是现代密码学发展的重要里程碑。从密码学产生至今，几乎所有的传统密码系统都是基于替换和置换这些初等方法。公钥密码学的出现使密码学的研究发生了巨大变化。与传统加密系统不同的是，使用这种方法的加密系统不只是公开加密算法本身，也公开了加密用的密钥。公钥算法是基于数学难题而不是基于替换和置换。更重要的是，与只使用一个密钥的对称传统密码不同，公钥密码学是非对称的，使用两个不同的密钥，这使得公钥密码技术具有与传统密码加密技术不同的应用特性，有效地解决了网络通信中的密钥分发交换问题。

#### 4.3.1 密码算法简介

加密解密是密码算法的基本技术，加 / 解密系统中的基本概念可以从下述简化的加密模型中导出，如图 4-4 所示。

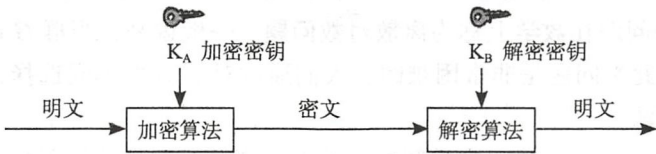


图 4-4 简化的加密模型

从图 4-4 中可以看出，明文输入在加密密钥  $K_A$  的作用下，通过加密算法转换成了密文，于是发送方可以通过公共信道（如互联网等）将密文发送给接收方；接收方则可以通过



解密算法以及密钥  $K_B$  将密文恢复成明文。而围绕着  $K_A$  和  $K_B$  是否相等, 加解密算法可以分为两大类, 一类叫对称加密, 另一类叫非对称加密。

对称加密是最基本的加密算法, 如现在广泛使用的 AES 算法就是对称加密算法。对称加密中的“对称”指的是加密过程和解密过程所用的密钥是相同的, 或者可以很容易地相互推导出来。很多对称加密算法就连算法本身都是对称的, 即完全一样的算法(甚至完全一样的模式)同时用于加密和解密。

非对称加密中加密和解密的密钥是不同的。一般来说, 非对称加密算法中的两个密钥, 一个称为公开密钥(公钥), 另一个称为私有密钥(私钥)。公钥必须公之于众, 而私钥必须秘密保存, 而由公钥导出私钥是困难的。

由于非对称加密的效率远低于对称加密, 因此实际中常用的模式是使用非对称算法为通信双方传输或协商一个会话密钥, 然后双方通过这个密钥使用性能更高的对称加密算法来进行通信。

### 4.3.2 公钥密码算法

1976 年 Diffie 和 Hellman 的《密码学的新方向》一文导致了密码学的一场革命。他们首次证明了在发送端和接收端无密钥传输的保密信息是可能的, 从而开创了公钥密码学的新纪元。

公钥密码算法是基于数学难题而非基于传统的替换和置换方法, 目前的公钥密码系统主要依赖于下面两类数学难题:

□ 大整数因子分解问题;

□ 离散对数问题。

第一类公钥密码算法是由 Rivest、Shamir、Adleman 提出的, 简称为 RSA 算法。它的安全性是基于大整数因子分解的困难性, 而大整数因子分解问题是数学上的著名难题, 因而可以确保 RSA 算法的安全性。RSA 算法是公钥密码算法中最具有典型意义的方法, 是人们研究的热点。对于密码破译者来说, 想已知密文推出明文就必须进行大整数的因子分解。

第二类公钥密码算法的安全性依赖于离散对数的计算困难性(简称 DLP 问题)。设  $G$  为一个有限 Abel 加法群, 假定  $g$  为  $G$  的某个生成元,  $a$  为任意的整数, 如果已知  $g$  与  $g^a$ , 如何求出整数  $a$  的问题在数学上称为离散对数问题。一般说来, 当群  $G$  选择得当, 且整数  $a$  充分大时, 求解此类问题是非常困难的。人们通过对群  $G$  的不同选择, 可以构造出各种不同的公钥密码算法。

椭圆曲线公钥密码算法建立在椭圆曲线的点的离散对数的计算困难性问题上。椭圆曲线密码算法使用了在有限 Abel 群 ( $GF(p)$  或者  $GF(2^m)$ ) 上构造的椭圆曲线。注意到两个椭圆曲线上的点相加所得到的点依然在原椭圆曲线上, 若  $P$  为椭圆曲线上的一个点, 在等式  $kP = P + P + \dots + P = Q$  中, 已知  $k$  和点  $P$  求点  $Q$  比较容易, 反之已知点  $Q$  和点  $P$  求  $k$  却是相当

困难的, 椭圆曲线密码体制正是利用这个困难问题设计而来, 例如, 可以使用  $Q$  为公钥, 而  $k$  为私钥。

椭圆曲线密码算法具有下面两个明显的优点: ① 能用较短的密钥长度实现和其他方案 (比如整数分解) 相同的安全级别; ② 所有的用户可以选择同一基域上的不同椭圆曲线, 使所有的用户使用同样的操作完成域运算。

### 4.3.3 区块链中使用的椭圆曲线

比特币区块链中采用了椭圆曲线公钥密码技术, 其所使用的椭圆曲线为 Certicom 推荐的 secp256k1 椭圆曲线。

Certicom 是国际上最著名的椭圆曲线密码技术公司, 已授权 300 多家企业使用 ECC 密码技术, secp256k1 为基于  $F_p$  有限域上的椭圆曲线, 由于其构造的特殊性, 其优化后的实现比其他曲线性能上可以提高 30%。

secp256k1 形如  $y^2 = x^3 + ax + b$ , 由六元组  $D = (p, a, b, G, n, h)$  定义:

□ 素数  $p = 2^{256} - 2^{32} - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1$ ;

□ 参数  $a$  和  $b$  分别为 0 和 7;

□  $G$  为该曲线的基点, 其压缩形式表示为 02 79BE667E F9DCBBAC 55A06295 CE870B07

029BFCDB 2DCE28D9 59F2815B 16F81798,

非压缩形式表示为 04 79BE667E F9DCBBAC

55A06295 CE870B07 029BFCDB 2DCE28D9

59F2815B 16F81798 483ADA77 26A3C465

5DA4FBFC 0E1108A8 FD17B448 A6855419

9C47D08F FB10D4B8;

□  $n$  为  $G$  的阶,  $n = \text{FFFFFFFF FFFFFFFF FFFFFFFF}$

FFFFFFFFE BAAEDCE6 AF48A03B BFD25E8C

D0364141;

□ 协因子  $h = 1$ 。

图 4-5 展示了实数域上的 secp256k1 椭圆曲线。

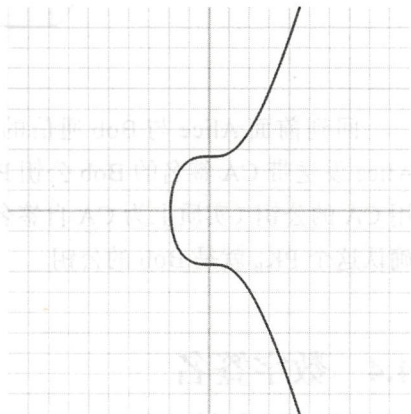


图 4-5 实数域上的 secp256k1 椭圆曲线

### 4.3.4 数字证书

依据公钥密码算法, 用户的公钥必须公开, 然而如何确定一个公开的密钥就是给定用户的公钥是一个必须解决的问题。例如, 在 Alice 与 Bob 进行的通信中, Alice 需要 Bob 提供他的公钥  $PK_B$ 。当 Alice 收到这个  $PK_B$  时, 她怎么才能确定这个  $PK_B$  就是 Bob 的公钥呢? 一个简单的解决方案就是使用数字证书。

数字证书是一个证书权威机构 (CA) 对一个实体所拥有的公钥的认证证书。CA 是这个



证书的签发机构，为通信双方所共同信任，作为受信任的第三方，承担公钥体系中公钥的合法性检验的责任。一个最简单的证书包含实体名称、实体的公开密钥以及 CA 的数字签名。目前大量采用的数字证书格式遵循 X.509 国际标准。

图 4-6 展示了 X.509 数字证书的基本结构，其中重要的内容包括证书所使用的签名算法、证书拥有者的名称、证书主体的公开密钥以及证书颁发者对证书的签名。

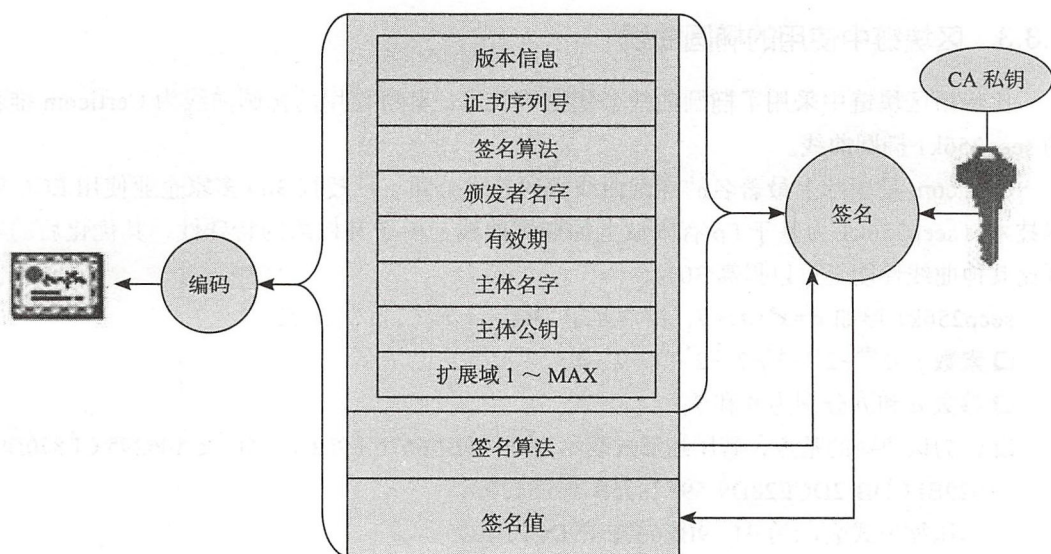


图 4-6 数字证书结构

回到前面 Alice 与 Bob 通信的例子。当 Alice 要求 Bob 提供他的公钥  $PK_B$  后，Bob 向 Alice 发送带 CA 签名的 Bob 公钥  $PK_B$  的数字证书。由于 Alice 也信任 CA，因此 Alice 可以用 CA 的公钥（实质上为 CA 自签名证书上的 CA 公钥）验证签名，一旦验证通过，Alice 就确认这个  $PK_B$  就是 Bob 的公钥。

## 4.4 数字签名

数字签名是公钥密码体制的一个重要应用模式。在双方通信过程中，发送方用自己的私钥加密需传输的信息形成数字签名后，连同原始信息发给接收方，接收方使用发送方的公钥验证签名，以确认所收到的信息就是发送方发送的信息。

### 4.4.1 数字签名简介

数字签名是以电子形式存在于数据信息之中的，或作为其附件或逻辑上与之有联系的数据，可用于辨别数据签署人的身份，并表明签署人对数据信息中所含信息的认可。

在形式上,一个数字签名方案通常包含三个多项式时间算法:  $(G, S, V)$ , 其中, 密钥生成算法  $G$  从可选私钥集中随机选择一个私钥, 算法输出私钥及对应的公钥; 签名算法  $S$  依据给定的消息和私钥, 生成一个签名; 签名验证算法则在给定消息、公钥和签名的基础上, 验证消息是否为指定签名者所签名。为了减少原始消息可能带来的巨大签名计算量, 通常是先生成原始消息的哈希值, 然后再对这个哈希值进行签名。

#### 4.4.2 数字签名标准与 ECDSA 算法

1991 年 8 月, 美国国家标准局 (NIST) 公布了数字签名标准 (Digital Signature Standard, DSS), 在标准中, 采用了 DSA (Digital Signature Algorithm) 算法。DSA 算法是 ElGamal 签名算法的变种, 其安全性基于离散对数难题。

ECDSA (Elliptic Curve Digital Signature Algorithm) 则是 DSA 算法的一个变种。与 DSA 算法使用的整数有限域不同, ECDSA 使用的是椭圆曲线群。区块链中应用的数字签名算法就是 ECDSA 算法。

##### 1. 椭圆曲线数字签名生成

假定 Alice 希望对消息  $m$  进行签名, 她所采用的椭圆曲线参数为  $D=(p, a, b, G, n, h)$  (例如, 可以采用前述的 secp256k1 曲线), 对应的密钥对为  $(k, Q)$ , 其中  $Q$  为公钥,  $k$  为私钥。Alice 将按如下步骤进行签名。

- 第 1 步 产生一个随机数  $d$ ,  $1 \leq d \leq n-1$ 。
- 第 2 步 计算  $dG=(x_1, y_1)$ , 将  $x_1$  转化为整数  $\overline{x_1}$ 。
- 第 3 步 计算  $r=\overline{x_1} \bmod n$ , 若  $r=0$  转向第 1 步。
- 第 4 步 计算  $d^{-1} \bmod n$ 。
- 第 5 步 计算哈希值  $\text{SHA}(m)$ , 并将得到的比特串转化为整数  $e$ 。
- 第 6 步 计算  $s=d^{-1}(e+kr) \bmod n$ , 若  $s=0$ , 则转向第 1 步。
- 第 7 步  $(r, s)$  即为 Alice 对消息  $m$  的签名。

##### 2. 椭圆曲线签名验证

为验证 Alice 对消息  $m$  的签名  $(r, s)$ , Bob 可以得到 Alice 所用的椭圆曲线参数以及 Alice 的公钥  $Q$ 。Bob 将按以下步骤操作。

- 第 1 步 验证  $r$  和  $s$  是区间  $[1, n-1]$  上的整数。
- 第 2 步 计算  $\text{SHA}(m)$  并将其转化为整数  $e$ 。
- 第 3 步 计算  $w=s^{-1} \bmod n$ 。
- 第 4 步 计算  $u_1=ew \bmod n$  以及  $u_2=rw \bmod n$ 。
- 第 5 步 计算  $X=u_1G+u_2Q$ 。
- 第 6 步 若  $X=O$ , 则拒绝签名, 否则将  $X$  的  $x$  坐标  $x_1$  转化为整数  $\overline{x_1}$ , 并计算  $v=\overline{x_1} \bmod n$ 。



第 7 步 当且仅当  $v=r$  时, 签名通过验证。

为具体说明椭圆曲线签名和验证算法的过程, 我们来看一个简化的例子: Alice 决定把 10 个比特币支付给 Bob, 矿工 Miner 负责把这笔账记录下来。这个过程是怎么使用签名和验证算法进行的呢?

首先 Alice 从自己钱包中取出 10 个比特币, 要将这 10 个比特币支付给 Bob, 于是交易消息  $m$  “Alice 支付 10 BTC 给 Bob” 产生了。这个消息会向全网广播。收到这个交易消息的用户会产生疑问: 这个交易是不是真的? 为打消其他用户的疑虑, Alice 需要对这段交易消息进行数字签名, 以向大家确定这个交易确实是由她发出的。为此, Alice 使用了 secp256k1 椭圆曲线进行数字签名, 这个对交易消息内容的签名需要使用 Alice 的私钥  $k$  (签名算法的第 6 步)。考虑到消息的规模和公钥密码算法的效率, 对交易消息进行的签名实际上是对交易消息的哈希值进行签名。由于密码哈希函数的抗碰撞性, 可以认为这样的转化是合理有效的。于是 Alice 向全网广播的内容除了交易消息本身外, 还包含 Alice 对消息的签名以及 Alice 的公钥信息。

假定 Alice 发送的交易消息连同签名发出后, 为矿工 Miner 所接收。为在区块链中记录这一交易, Miner 首先需要验证这个交易是不是 Alice 发出的, 即进行签名验证的工作。为此, Miner 也使用了同样的 secp256k1 椭圆曲线来验证签名。对 Alice 签名验证的过程需要使用 Alice 的公钥参与, 如签名验证算法中的第 5 步就使用了 Alice 的公钥  $Q$ 。如一切顺利的话, Miner 可以验证交易消息  $m$  “Alice 支付 10 BTC 给 Bob” 确实是 Alice 发出的, Miner 可以在之后的操作中把这个交易记入区块链中。如果签名验证失败, 表明 Miner 收到的这个消息存在问题, Miner 会放弃将相关的交易记入区块链的操作。

利用椭圆曲线的签名和验证算法, 一方面可以保证用户的账户不被冒名顶替, 另一方面也能确保用户不能否认其所签名的交易。用户发起交易的时候, 使用自己的私钥对交易信息签名, 矿工收到信息后用户的公钥对签名进行验证, 一旦通过, 该交易信息就可通过矿工进行记账, 最终完成交易。

#### 4.4.3 其他的数字签名方法

作为一项重要的计算机安全技术, 数字签名的基本作用是保证传送的信息不被篡改和伪造, 并确认签名者的身份。实际上, 人们还提出了许多特殊的签名机制, 以应对不同的应用场合。

“群签名”在 1991 年提出, 其基本思想是一群人中的任何一个人可生成一群签名, 外界可验证其合法性, 即此签名的确为此群中的某人生成, 但不能确定到底是他们当中哪一位签的 (隐私保护), 在发生争议的情况下, 可由一拥有特权的群管理员 “打开” 争议的签名找出真正的签名者 (可追踪)。

2001 年, “环签名”提出。环签名可以被视为一种特殊的群签名。与已有的群签名相比, 环签名是完全匿名的, 且成员在生成群签名时可自由地选择其他成员构成群。环签名

没有可信中心,没有群的建立过程,对于验证者来说签名者是完全匿名的。环签名的这种无条件匿名性在对信息需要长期保护的一些特殊环境中非常有用。例如,即使底层困难问题被攻破也必须保护匿名性的场合。

1982年提出的“盲签名”具有盲性这一特点,可以有效保护所签署消息的具体内容,所以在电子商务和电子选举等领域有着广泛的应用。盲签名允许消息者先将消息盲化,而后让签名者对盲化的消息进行签名,最后消息拥有者对签字除去盲因子,得到签名者关于原消息的签名。对于盲签名算法,消息的内容对签名者是不可见的,而且在签名被接受者公开后,签名者不能追踪签名。

在数字签名过程中,有时需要多个用户对同一消息进行签名和验证。能够实现多个用户对同一消息进行签名的数字签名称为多重数字签名。

根据签名过程的不同,多重数字签名方案可分为两类:一类为有序多重数字签名方案,另一类为广播多重数字签名方案。两类方案都包含消息发送者、消息签名者和签名验证者,在广播签名方案中还包括签名收集者。

在有序多重数字签名方案中,由消息发送者规定消息签名顺序,然后将消息发送到第一个签名者。除了第一个签名者外,每一位签名者收到签名消息后,首先验证上一签名的有效性。如果签名有效,继续签名,然后将签名消息发送到下一个签名者;如果签名无效,拒绝对消息签名,终止整个签名。当签名验证者收到签名消息后,验证签名的有效性,如果有效,多重签名有效,否则多重签名无效。

在广播多重数字签名中,消息发送者同时将消息发送给每一位签名者进行签名,然后签名者将签名消息发送到签名收集者,由收集者对签名消息进行整理并发送给签名验证者,签名验证者验证多重签名的有效性。

## 4.5 零知识证明

零知识证明质是一种涉及两方或更多方的协议。证明者向验证者证明并使其相信自己知道或拥有某一消息,但证明过程不能向验证者泄漏任何关于被证明消息的信息。在区块链中,这些消息可以指交易信息数据。

零知识证明并不保证这个证明泄漏的信息为零,因为它证明了它要证明的论断是正确的,而这个论断本身可能已经泄漏的信息。因此能做到最好的只是泄漏尽可能少的信息。零知识证明一开始也被更贴切地叫作“最小透露的证明”(minimum disclosure proof)。了解在具体场景里零知识证明意味着泄漏多少信息是非常重要的。

经典的零知识证明是交互式的,直接用这些交互式证明不太实际。因此实际应用中需要很多优化。优化中需要引入一些并没有被证明的(也没有攻破的)新的安全假设。简明非交互零知识证明(zero-knowledge Succinct Non-interactive Arguments of Knowledge, zkSNARKs)是成功应用于区块链中的零知识证明技术,可在无须泄漏交易数据的情况下,



向验证者证明某个交易是合法的。Zcash 使用了 zkSNARKs，以太坊和 Polkadot 也计划采用 zkSNARKs。

为应用 zkSNARKs 技术，以 Zcash 为例，首先需要将交易验证函数转换为称为“算术电路”的基本逻辑操作，即由加减乘除这些基本算术构成的操作，接着通过头号排名约束系统（Rank 1 Constraint System, R1CS），检测算术门（如加法门、乘法门等）计算的正确性。在 R1CS 表示中，验证者需要检查许多约束，Zcash 采用了二次型算术程序（Quadratic Arithmetic Program, QAP）的电路表示方法，使得单个约束需要检测的是多项式而非数字。尽管多项式可能很庞大，但一旦多项式间的一致性不保持，在多个点也不可能保持。因此，只需要检查两个多项式是否在一个随机选择的点匹配，就可以很高的概率验证证明。zkSNARKs 使用了同态加密和椭圆曲线配对的数学工具，使得证明者难以提前知道验证者选择的检测点。实际上，上述数学工具的使用，使得无论证明者还是验证者都不能知道检测点的精确值。同时对原始多项式的随机移位，使得无论证明者既保持其输入值的保密性，也满足多项式一致性的验证。

## 4.6 区块链中的隐私问题

基于公有链的区块链是一个公开数据库，每个账户里的资产、交易记录都是公开的。这样的公开透明带来了一个隐私问题：一旦确定账户地址，任何一个用户都能通过区块链查询到这个账户里的所有资产和历史交易。一种解决方案就是建议用户每次收款都使用新地址，这样交易方就只能看到该笔交易，而不是全部。然而，多种原因会导致用户不得不重复使用同一个账户，这使得基于用户行为的链接攻击成为可能。

比特币系统在设计时考虑到了这个问题，提供了一定程度上的隐私保护机制。为避免由于比特币系统公开交易金额而泄漏用户的身份隐私，比特币采用一次性公钥，用公钥作为用户假名和账号。这样，由于用户每次交易可以采用不同的假名，因而比特币具有一定的隐私保护能力。然而，随着比特币从虚拟货币向实物货币的转化，当用户需要利用比特币购物时，其付款所用假名或账号立即将用户的个人信息泄漏了。同样，当用户公开自己的比特币假名以接受付款或捐款时，其在比特币系统中的真实物理身份实际上也暴露了。实际上，电子钱包或比特币交易所的实名认证、同笔交易中多个输入间的关联关系、网络报文中 IP 地址与比特币地址的对应关系等都可能泄漏用户的真实身份。

比特币用户隐私泄漏主要有两个方面的原因：一是公开的交易额、交易元数据和全网账本使得攻击者可以提取有关用户身份特征的大量信息；二是交易中付款账号与收款账号明确关联的特征使得攻击者可以利用链接攻击追踪整个历史交易路径，结合比特币作为实物货币时的真实身份泄漏，系统的隐私保护能力被严重削弱。

区块链隐私保护既需掩盖交易细节，又需验证交易的正确性。区块链目前的隐私保护方案包括混币、环签名、同态加密、零知识证明等方式。

### (1) 混币

混币通过割裂输入地址和输出地址之间的关系实现匿名。如果在一个多人参与的交易中,包括大量输入和输出,试图在输入和输出中找出每个人的对应并非一件容易的事情,于是,输入与输出之间的联系被事实上割裂。Mixcoin 协议试图打乱支付账号和收款账号之间的关系,从而提高比特币及类似系统的匿名性。CoinJoin 技术是一种混淆比特币交易数据中用户敏感信息关联性的比特币交易压缩方法。一个 CoinJoin 交易是指:多个用户同意一个单独的交易,这项交易有多项相同大小的输出。交易中每一个输入的签名对每个用户是独立的,用户可以通过协商允许多个输入和多个输出的交易,然后每个用户独立分散地签名,最后将签名合并允许交易。这样,一个普通的区块链观察人员无法分辨输出和用户的对应关系。达世币则通过建立 Darksend 节点,将多笔交易拼凑成一笔交易一起执行,以打断发送地址和接收地址之间的联系,实现数字货币交易模糊化。

### (2) 环签名

环签名是一类简化的类群签名。环中一个成员利用他的私钥和其他成员的公钥进行签名,但却不需要征得其他成员的允许,而验证者只知道签名来自这个环,但不知道谁是真正的签名者。环签名可以解决对签名者的匿名问题,允许一个成员代表一组人进行签名而不泄漏签名者的信息。比特币是最早引入环签名和隐身地址的加密货币。暗网币则在比特币基础之上进行开发 and 创新的。暗网币的环签名是块链上的混币服务。这种混币具有相同金额的输入,并且使用了多个别人的公钥,只知道是从这一群人中的一个发送的,但无法判断是哪一个,也无法通过金额分析来判断输入输出对。门罗币应用了环签名,用发送者的私钥和多个无关者的公钥加密交易数据,再用所有人的公钥解密数据,以此隐藏交易的发送者。

### (3) 同态加密

同态加密是一种无须对加密数据进行提前解密就可以执行计算的方法。通过在区块链上使用同态加密技术存储数据可以达到一种完美的平衡,不会对区块链属性造成任何重大的改变。也就是说,区块链仍旧是公有区块链,但区块链上的数据将会被加密。

### (4) 零知识证明

零知识证明支持在无须泄漏数据本身的情况下证明某些数据运算,允许验证者可以验证证明而不需要证明,无须泄漏除了“它是真实的”之外的任何信息。零币是一种去中心化的混币技术,把一定数量如 1 个比特币转化为零币,采用零知识证明技术花费零币或转化为其他零币,使得在零币之中的交易不可追踪。Zcash 应用了 zkSNARKs 零知识证明实现方案,可在无须泄漏交易数据、无任何额外信息的前提下,向验证者证明某个交易是正确的。Zcash 实现了交易关联性和转账金额的隐藏,在无须泄漏交易数据的前提下,完成交易的确认和验证。通过扩展 Zcash,可以实现具有隐私保护的智能合约,可支持任意交易的隐私保护。同态加密基于同态映射保证了先运算后加密和先加密后运算的结果相同,从而可基于加密数据实现交易验证。

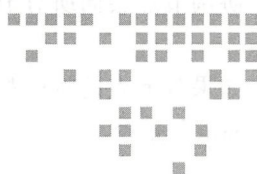


## 4.7 小结

区块链中大量使用了现代密码技术。密码学哈希函数被用来验证区块链所存储信息的完整性，同时也被用于设计基于工作量证明的共识技术；公钥密码技术被广泛应用于区块链用户的身份认证，尤其是数字签名技术在账本信息的构造中保证了信息的不可否认性；零知识证明及环签名等技术被用于保护用户交易行为的隐私。

### 习题

- [1] 编写一个程序，验证密码学哈希函数生成给定前缀的哈希值的困难性。
- [2] 编写程序，构造一个简单的基于区块链的数据存储系统，实证这个系统的防篡改特性。



# 共识算法

作者：邹 均

第4章介绍了作为区块链核心技术的密码学的一些重要知识，在本章中将为读者介绍区块链技术的另一个重要核心技术——分布式共识算法。分布式共识算法是分布式系统中保证系统状态一致性的重要技术，是分布式文件系统、分布式数据库的重要基础。区块链系统采用分布式共识算法在无中心节点控制，又可能存在破坏节点的环境下确立系统状态，从而建立信任。区块链因此也被称为“信任机器”<sup>①</sup>。由此可见，共识算法是区块链技术中需要重点掌握的关键技术。共识算法因不同的应用场景而有所不同。从区块链的部署架构来看，有适合于公有链的共识算法，也有适合于联盟链或私有链的共识算法。公有链上的共识算法一般需要支持高扩展性，能够在共识节点动态出入网络的情况下保证共识流程的节奏，同时又要防止拜占庭节点对网络的攻击。受限于FLP定理和CAP定理，公有链的共识算法一般不能保证强一致性。而另一方面，联盟链的共识算法一般需要支持强一致性、高性能。但对高扩展性和防拜占庭节点攻击方面的要求往往没有公有链的要求那么高。

不同的共识算法在一致性、正确性、可终止性、交易吞吐量、记账频率、扩展性和安全性上各有不同。在实际应用中，需要根据应用场景，选择能满足应用需求的共识算法。下面我们来介绍分布式共识算法的背景和典型的共识算法。

## 5.1 分布式共识算法背景

在数据库系统中，两阶段提交（Two-Phase Commit）是一个普遍使用的保证一致性的方法。如图5-1所示，两阶段提交由一个协调节点（Coordinator）居中控制。

<sup>①</sup> 《经济学人》2015年10月。



第一阶段，协调节点会向所有节点发送一个询问“是否准备好”的消息，各节点会响应该消息；

第二阶段，如果各节点的响应都是准备好的回复，协调节点会向各节点发“提交”指令，否则会发“回滚”指令。

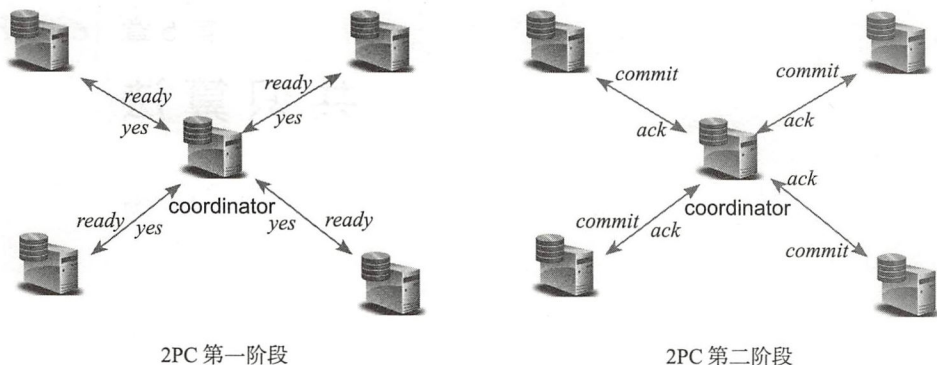


图 5-1 两阶段提交协议

这样所有节点的状态都会保持一致，要么提交新的状态，要么回滚成以前的状态。两阶段提交协议在没有出现节点故障的情况下可以保证系统状态的一致性，但是当协调节点和另外一个节点同时出故障的时候，系统没有办法知道故障节点的状态，因此无法决定是提交还是回滚。

为了解决这个问题，三阶段提交（Three-Phase Commit）协议被提出来<sup>[1]</sup>（见图 5-2），即在正式提交前，协调节点再发一次“准备提交”的消息，在收到所有节点的回复后，协调节点才发“提交”的指令。三阶段提交能部分解决两阶段提交中协调节点和另一节点同时出故障的问题，但是系统过分依赖一个协调节点，另外其余节点也没有办法形成对协调节点是否出现故障的共识，因此，无论两阶段提交还是三阶段提交，都不能作为一个分布式共识算法，只能在通常情况下保障分布式系统的一致性。



图 5-2 三阶段提交协议

两阶段提交和三阶段提交多数用在同步通信的分布式系统中，而通常的分布式系统是

由多个主机通过异步通信方式组成网络集群。在这样的一个异步系统中,需要主机之间进行状态复制,以保证每个主机达成一致的状态共识。然而,异步系统中可能出现无法通信的故障主机,而主机的性能可能下降、网络可能拥塞,这些可能导致错误信息在系统内传播。因此需要在默认不可靠的异步网络中建立容错协议,以确保各主机达成安全可靠的状态共识。

利用区块链构造基于互联网的去中心化账本,需要解决的首要问题是如何实现不同账本节点上账本数据的一致性和正确性。这就需要借鉴已有的、在分布式系统中实现状态共识的算法,确定网络中选择记账节点的机制,以及如何保障账本数据在全网中形成正确、一致的共识。

在20世纪80年代出现的分布式系统共识算法,是区块链共识算法的基础。下面我们从基本的拜占庭将军问题入手,逐步介绍适合于联盟链/私有链和公有链的共识算法。

### 5.1.1 拜占庭将军问题

拜占庭将军问题是区块链平台的共识机制需要解决的一个核心问题。学习共识算法首先要理解拜占庭将军问题。

#### 1. 拜占庭将军问题提出与约束条件

拜占庭将军问题是 Leslie Lamport 在20世纪80年代提出的一个假象问题<sup>[2]</sup>。拜占庭是东罗马帝国的首都,由于当时拜占庭罗马帝国国土辽阔,每支军队的驻地分隔很远,将军们只能靠信使传递消息。发生战争时,将军们必须制订统一的行动计划。然而,这些将军中有叛徒。叛徒希望通过发布错误消息来影响忠诚的将军们制定统一的行动计划,以达到破坏忠诚的将军们统一行动的目的。因此,将军们必须有一个预定的方法协议,满足两个条件:

- A. 使所有忠诚的将军能够达成一致;
- B. 少数几个叛徒不能使忠诚的将军做出错误的计划。

条件B比较难用形式化方式描述,这里需要了解的是将军们怎么做决策。一般来说,每个将军会把他观察到的敌情派通信员告知其他将军。假设  $v(i)$  是第  $i$  个将军通知其他将军的消息,每个将军会采用某种方法把所有收到的消息  $v(1) \dots v(n)$  组成一个单一的行动计划,其中  $n$  是将军的数目。条件A可以通过采用同一方法组成行动计划来达到,而条件B则可以通过某种鲁棒性方式来达到,例如,通过多数人的意见来决定最后的行动计划。

如果要满足条件A,下列条件必须满足:

- 1) 每个忠诚的将军必须收到相同的命令值  $v(1), \dots, v(n)$ 。(  $v(i)$  是第  $i$  个将军的命令);
- 2) 如果第  $i$  个将军是忠诚的,那么他发送的命令和每个忠诚将军收到的  $v(i)$  相同。

Lamport 把以上所有将军互相发命令的问题简化成一个将军发指令给  $n-1$  个副官,从而得出拜占庭将军问题。



拜占庭将军问题——一个发送命令的指挥官要发一个命令给其余  $n-1$  个副官，使得：

IC1. 所有忠诚的接收命令的副官遵守相同的命令；

IC2. 如果发送命令的指挥官是忠诚的，那么所有忠诚的接收命令的副官将遵守所接收的命令。

拜占庭将军论断——Lamport 指出在将军通过通信员口头传递信息的情况下，只要有  $1/3$  以上的叛徒，则没法保证忠诚的副官能达成一致的行动。

Lamport 采用一个简单的图来说明在一个指挥官两个副官的情况下，只要有一个叛徒存在，就不可能达成共识。如图 5-3 所示，深灰色表示叛徒，当指挥官是叛徒的情况下，忠诚的副官 1 和副官 2 没法遵从同样的命令，因此没办法保障 IC1 的成立。

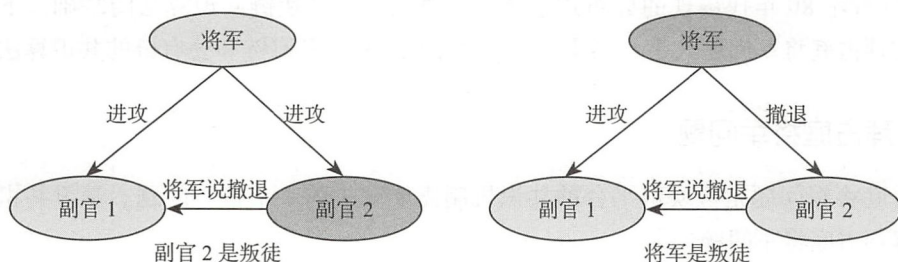


图 5-3 三个拜占庭将军，一个叛徒

Lamport 对拜占庭将军问题的研究表明，当  $n > 3m$  时，即叛徒的个数  $m$  小于将军总数  $n$  的  $1/3$  时，通过口头同步通信（假设通信是可靠的），可以构造同时满足 IC1 和 IC2 的解决方案，即将军们可以达成一致的命令。但如果通信是可认证、防篡改伪造的（如采用 PKI 认证，消息签名等），则在任意多的叛徒（至少得有两个忠诚将军）的情况下都可以找到解决方案。

Lamport 对口头同步通信的定义如下：

- A1. 每个发送的消息都会被正确接收；
- A2. 接收消息的人知道是谁发送的消息；
- A3. 消息没有发送可以被检测到。

Lamport 利用递归的方式定义了一个能在系统中叛徒少于  $1/3$  的将军总数的情况下，采用口头同步信息达成共识的算法，简单表示成  $OM(m)$ ，即在最多  $m$  个叛徒的情况下，该算法能使总共超过  $3m+1$  的将军解决拜占庭将军的问题。该算法假设有一个 majority 的多数函数，当大多数的  $v_i$  值等于  $v$ ，则  $\text{majority}(v_1, \dots, v_{n-1})$  等于  $v$ 。如果  $v_i$  不存在，该函数值是 RETREAT（撤退是缺省值）。

## 2. 算法与定理

### (1) 算法 $OM(0)$ ( $m=0$ ，没有叛徒的情况)

- ① 指挥官发他的值（相当于命令）给每一个副官。

② 每个副官用他接收到指挥官的值作为当前值, 如果没有接收到, 用 RETREAT (缺省是撤退)。

## (2) 算法 OM ( $m$ ), $m > 0$

① 将军发他的值给每一个副官。

② 对每个  $i$ , 让  $v_i$  等于副官  $i$  从指挥官那里接收到的命令, 或者如果没有接收到指令的话值就是 RETREAT (缺省撤退)。副官  $i$  作为算法 OM( $m-1$ ) 的将军, 发  $v_i$  给其他  $n-2$  个副官。

③ 对每个  $i$ , 以及每个  $j \neq i$ , 让  $v_j$  等于第 2 步中采用算法 OM( $m-1$ ) 副官  $i$  从副官  $j$  接收到的值, 或者没有收到值则缺省为 RETREAT。副官  $i$  取  $\text{majority}(v_1, \dots, v_{n-1})$  的值。

为了证明算法 OM ( $m$ ) 对任意  $m$  都能解决拜占庭将军问题, 先证明以下引理:

**引理 1:** 对任意  $m$  和  $k$ , 如果有多于  $2k+m$  个将军和最多  $k$  个叛徒, 算法 OM ( $m$ ) 满足 IC2。

**证明:** 证明采用归纳法。IC2 只是规定在指挥官是忠诚的情况下需要满足的条件。根据 A1, 可以很容易看出当指挥官是忠诚的时候, OM (0) 成立, 所以引理 1 在  $m=0$  的情况下成立。我们现在假设在  $m-1$ ,  $m > 0$  的情况下成立, 我们只需要证明  $m$  情况下也成立, 通过数学归纳法, 我们可以得出引理 1 成立。在算法 OM ( $m$ ) 第一步, 忠诚的指挥官发一个值  $v$  给所有  $n-1$  个副官。在第二步, 每个忠诚的副官递归调用 OM ( $m-1$ ), 总共  $n-1$  个将军。

根据假设  $n > 2k+m$ , 我们可以得到  $n-1 > 2k+(m-1)$ , 所以我们可以采用归纳法来得出每个忠诚的副官从第  $j$  个忠诚的副官处得到  $v_j=v$ 。因为最多只有  $k$  个叛徒, 并且  $n-1 > 2k+(m-1) > 2k$ , 所以  $n-1$  个副官中的多数是忠诚的。因此, 每个忠诚的副官都在  $n-1$  个副官的多数中收到  $v_i=v$ , 所以他在第三步会得到  $\text{majority}(v_1, \dots, v_{n-1})=v$ , 证明了 IC2, 即当指挥官是忠诚的情况下, 所有忠诚的副官都执行接收的命令。

以下定理确认算法 OM ( $m$ ) 能解决拜占庭将军问题。

**定理 1:** 对任意  $m$ , 如果在至多  $m$  个叛徒而将军数超过  $3m$  个的情况下, 算法 OM ( $m$ ) 满足条件 IC1 和 IC2。

**证明:** 采用归纳法来证明。如果没有叛徒, 很容易得出 OM (0) 满足 IC1 和 IC2。我们假设定理在 OM ( $m-1$ ) 时成立, 然后我们只需要证明在 OM ( $m$ ),  $m > 0$  时也成立。

首先我们考虑将军是忠诚的情况。在引理 1 中, 使  $k=m$ , 我们可以得出 OM( $m$ ) 满足 IC2。如果将军是忠诚的, IC1 也会自动满足。所以我们只需要在将军是叛徒的情况下验证 IC1。最多只有  $m$  个叛徒, 由于指挥官是其中一个, 因此有  $m-1$  个副官是叛徒。因为有超过  $3m$  个将军, 会有超过  $3m-1$  个副官, 而且  $3m-1 > 3(m-1)$ 。我们因此采用递归来得出 OM ( $m-1$ ) 满足条件 IC1 和 IC2。所以, 对每个  $j$ , 任何两个忠诚的副官在第三步的时候都会得到相同的  $v_i$ 。(如果两个副官中的一个副官  $j$ , 可以从 IC2 中得出, 否则可以从 IC1 中得出。) 因此, 任意两个忠诚的副官都可以得到一样的向量  $v_1, \dots, v_{n-1}$ , 因此会在第三步获得一样的  $\text{majority}(v_1, \dots, v_{n-1})$ , 证明 IC1 成立。



综上所述,拜占庭将军问题的实质就是要寻找一个方法,使得在有故障的情况下能建立系统的共识。在分布式系统中,特别是在区块链网络中的环境也和拜占庭将军的环境类似,有运行正常的服务器(类似忠诚的拜占庭将军),有存在故障的服务器,还有破坏者的服务器(类似叛变的拜占庭将军)。共识算法的核心是在正常的节点间形成对网络状态的共识,也就是解决拜占庭将军问题。

### 5.1.2 共识系统的基本定义

从 5.1.1 节可见,拜占庭将军问题在一个分布式系统中是一个非常有挑战的问题。在这里,我们先给出分布式计算中有关拜占庭缺陷和故障的 4 个定义。

**定义 1: 拜占庭缺陷 (Byzantine Fault)。**任何从不同观察者角度看表现出不同症状的缺陷。

**定义 2: 拜占庭故障 (Byzantine Failure)。**在需要共识的系统中由于拜占庭缺陷导致丧失系统服务。

**定义 3: 宕机缺陷 (Fail-Stop Fault)。**在需要共识的系统中导致进程停止运行发生的缺陷;该缺陷不对系统产生其他副作用。

**定义 4: 宕机恢复故障 (Fail-Recovery Failure)。**在需要共识的系统中导致进程停止运行发生的故障;该故障在重启进程后恢复,不对系统产生其他副作用。

在分布式系统中,不是所有的缺陷或故障都能称为拜占庭缺陷或故障。像宕机、丢消息等缺陷或故障不能算作拜占庭缺陷或故障。拜占庭缺陷或故障是最严重缺陷或故障。拜占庭缺陷有不可预测、任意性的特点,例如遭黑客破坏。中木马的服务器就是一个拜占庭服务器的例子。

在一个分布式系统中,所有的进程都有一个初始值。在这种情况下,共识问题 (Consensus Problem) 就是要寻找一个算法和协议,使得该协议满足以下三个属性:

- 1) 一致性 (Agreement): 所有的非缺陷进程都必须同意同一个值。
- 2) 正确性 (Validity): 所有非缺陷的进程所同意的值必须来自非故障进程所提议的值。
- 3) 可结束性 (Termination): 每个非缺陷的进程必须最终确定一个值。

通常也把满足一致性和正确性称为安全性 (Safety),而满足可结束性称为活性 (Liveness)。

根据 Fischer-Lynch-Paterson 的理论 (见 5.1.3 节),在异步通信的分布式系统中,只要有一个拜占庭缺陷的进程,就不可能找到一个共识算法可同时满足上述要求的一致性、正确性和可结束性,也就是说,不能同时满足安全性和活性。

在实际情况下,根据不同的假设条件,有很多不同的共识算法被设计出来,这些算法各有优势和局限。算法的假设条件有以下几种情况。

- 1) 故障模型: 非拜占庭故障 / 拜占庭故障。
- 2) 通信类型: 同步 / 异步。

- 3) 通信网络连接: 节点间直连数。
- 4) 信息发送者身份: 实名 / 匿名。
- 5) 通信通道稳定性: 通道可靠 / 不可靠。
- 6) 消息认证性: 认证消息 / 非认证消息。

通常使用的共识算法, 如果按照严格的 FLP 异步通信的假设, 往往只满足安全性而牺牲活性。也就是说, 大部分使用的异步通信网络下的共识算法, 往往不能保证在有限的时间内总能达成共识。

### 5.1.3 Fisher-Lynch-Paterson 定理

1983 年, Fischer、Lynch 和 Paterson 三位科学家给出以下定理<sup>[3]</sup>:

**Fischer-Lynch-Paterson (简称 FLP) 定理:** 在一个多进程异步系统中, 只要有一个进程不可靠, 那么就不存在一个协议, 能保证有限时间内使所有进程达成一致。

FLP 定理不考虑拜占庭故障, 而只是一般的宕机故障。它假设异步通信是可靠的, 也就是说异步通信能成功传递消息, 而且能保证只传递一次消息而不会传递重复消息。在这里, 异步通信的意思是对进程处理速度和发送消息的延迟不作任何假设, 也就是说进程处理速度有可能非常慢, 消息的延迟也有可能非常大。同时假设进程没有一个同步的时钟可以访问, 也就是说不能依靠超时来做判断。最后, 也假设没办法通过监控来判断进程是否出现故障。也就是说, 没法分辨一个进程是出现故障还是运行得非常慢。

我们先给出 FLP 定理中共识协议的一些基本定义和假设。

**定义 1:** 一个共识协议  $P$  是一个有  $N$  个进程 ( $N \geq 2$ ) 的异步通信系统。每个进程  $p$  有一个一位的输入寄存器  $x_p$ , 一个输出寄存器  $y_p$ , 它们的取值范围是  $\{b, 0, 1\}$ 。每个进程有无限的内存。 $p$  的内部状态由它的输入 / 输出寄存器、内存、程序计数器决定。 $p$  是一个基于转换函数的确定性 (deterministic) 的进程。也就是说, 如果收到一个消息,  $p$  的状态就会转到一个确定的状态。初始状态 (initial state) 是不包含输入寄存器的所有内部状态。输出寄存器的值如果是 0 或者 1, 这个视为决定状态。如果进程  $p$  进入决定状态, 那么转换函数就不能再改变输出寄存器, 也就是说, 输出寄存器是仅“一次写”。当所有的非故障进程都进入一个相同值 (0 或 1) 的决定状态, 那么系统就达成了共识。整个  $P$  系统是由每个进程的转换函数和其相应的输入寄存器的初始值决定的。

**定义 2:** 系统中的进程通过互相发消息来进行通信。一个消息可以表示成  $(p, m)$ , 其中  $p$  是接收消息的进程,  $m$  是消息的值。 $m \in M$ ,  $M$  是一个固定的消息集合。消息在发送后并在没有接收前, 存储在消息缓存中。消息的操作支持两个抽象函数:

$\text{send}(p, m)$ : 把  $(p, m)$  放入消息缓存;

$\text{receive}(p)$ : 把  $(p, m)$  从消息缓存中删除, 同时返回  $m$ , 在这种情况下  $(p, m)$  已经送到。否则返回特殊的空集  $\emptyset$ , 同时不改缓存。

由此我们看到, 虽然每个进程的转换函数是决定性的, 但在异步通信环境下, 由于存



在不确定的延迟, 接收消息是非确定性的。也就是说, 即使有消息在缓存, 调用  $receive(p)$  也可能多次返回空集  $\emptyset$ 。

**定义 3:** 共识系统的配置 (configuration) 由所有进程的內部状态, 加上消息缓存组成。初始配置 (initial configuration) 是每个进程从初始状态开始, 同时消息缓存是清空的状态。

**定义 4:** 一个步骤 (step) 是一个单一进程的一次执行, 把一个配置状态转到另一个配置状态。假设  $C$  是一个配置, 一个步骤 (step) 的执行其实有两个阶段。第一阶段是进程  $p$  调用  $receive(p)$  接收信息  $m \in M \cup \{\emptyset\}$ 。然后取决于  $p$  的内部状态和  $m$ ,  $p$  进入一个新的内部状态, 并发有限的消息给其他进程。因为  $p$  是确定性的, 所以步骤完全取决于  $e=(p, m)$ 。 $e$  称为一个事件。 $e(C)$  表示结果配置, 同时我们称  $e$  可以作用于  $C$ 。需要了解的是事件  $(p, \emptyset)$  也总是可以作用于  $C$ , 所以总是有可能一个进程会走另一个步骤 (step)。

**定义 5:** 一个从  $C$  的计划 (schedule) 是一个可以作用于  $C$  的有限或无限的事件序列  $u$ 。相关的序列步骤叫作一个运行 (run)。如果  $u$  是有限的, 我们用  $a(C)$  来表示结果配置, 那么可以说结果配置可以由  $C$  到达 (reachable)。一个从初始配置 (initial configuration) 能到达的配置叫作可访问 (accessible)。以后, 所有提到的配置假设都可访问。

计划 (schedule) 有互换性 (commutativity), 见下面引理:

**引理 1:** 假设有一个配置  $C$ , 两个不同的计划  $\sigma_1$  和  $\sigma_2$  能分别作用于  $C$ , 并分别得到结果  $C_1$  和  $C_2$ 。如果在  $\sigma_1$  和  $\sigma_2$  没有相同的进程, 那么  $\sigma_2$  可以作用于  $C_1$ ,  $\sigma_1$  可以作用于  $C_2$ , 同时结果配置是一样的。

**证明:** 由于  $\sigma_1$  和  $\sigma_2$  没有重合的进程, 且每个进程都是确定性的, 其每一步骤只改变一个进程的內部状态, 而且改变的状态也是确定的。另外根据 FLP 的假设, 系统没有同步的时钟可以访问, 因此状态的改变与时间无关。因此  $\sigma_1 + \sigma_2$  的结果状态和  $\sigma_2 + \sigma_1$  的状态将一致, 如图 5-4 所示。

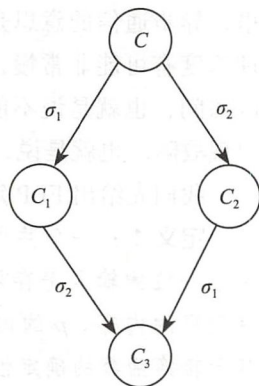


图 5-4 计划交换性

FLP 定理的作者采用反证法来证明 FLP 的正确性。他们首先假设在有一个进程出故障的情况下, 异步系统能够达成共识状态, 然后推出几个自相矛盾的引理。其中一个引理如下。

**引理 2:** 任何一个异步通信共识协议  $P$ , 都有一个共识状态不确定的初始配置 (initial configuration)。

这里所说的共识状态, 指的是系统中所有非故障进程的  $y$  寄存器都进入全是“0”或者全是“1”的状态。如果全是“0”, FLP 定理作者把该状态定义成“0-valent”, 如果全是“1”, 则是“1-valent”。不确定状态, 是指有些是“0”, 有些是“1”, 则定义成“bi-valent”, 也就是说无法达成共识。

FLP 定理的作者采用反证法证明引理 2。其证明方式比较抽象难懂。我们这里用通

俗的语言来描述其证明。首先假设协议  $P$ ，只有一个共识状态确定的初始配置，要么就是“0-valent”，要么就是“1-valent”。那么在所有可能的初始配置中，能够找到一个“0-valent”的初始配置，我们把它叫  $C_0$  的初始配置。 $C_0$  通过一些步骤最后进入“0-valent”的决定状态，同时找到另外一个除进程  $p$  之外状态和  $C_0$  完全一样的  $C_1$  初始配置，该配置通过一些步骤最后进入“1-valent”状态。 $C_0$  和  $C_1$  只是在进程  $p$  的状态不一样。如果进程  $p$  出现故障，也就是说进程  $p$  不能接收或发送消息，那么  $C_0$  和  $C_1$  进入决定状态所运行的步骤就会一样。也就是说，在一个系统中，存在着一个运行计划 (schedule)，有可能分别进入“0-valent”和“1-valent”的状态，因此和假设矛盾，从而证明了引理 2 的正确性。

在证明了引理 2 的基础上，FLP 定理作者更进一步证明了引理 3。

**引理 3：** 让  $C$  是  $P$  的一个不确定 (bi-valent) 配置，让  $e=(p, m)$  是一个可以作用于  $C$  的事件，让  $\zeta$  是一组不运行  $e$  而能从  $C$  达到的配置，同时让  $\partial=e(\zeta)=\{e(E)|E\in\zeta, \text{同时 } e \text{ 作用于 } E\}$ ，那么  $\partial$  包含一个不确定 (bi-valent) 的配置。

FLP 作者也是采用反证法来证明该引理，其证明过程比较烦琐，这里不做展开。其证明过程主要采用了引理 2 的证明技巧，同时加上引理 1，也即运行计划的可交换性来得出自相矛盾的结果。

在引理 2 和引理 3 正确的基础上，可以容易地用反证法来导出 FLP 定理的正确性。

FLP 定理是共识算法中一个非常重要的基础性定理。该定理的出现在共识算法领域具有里程碑式的意义，它推翻了过去很多研究者在共识算法领域的一些结论。在使用 FLP 定理的时候，我们一定要了解它的系统模型，也包括它的假设条件。在 FLP 的假设条件中，异步通信系统没有对进程接收消息、处理消息、回复消息的时间给出上限。所以无法判断一个进程是出了故障，还是因为要长时间处理而无回应。FLP 中的故障不是拜占庭故障，只是宕机类的故障。理解 FLP 定理的关键在于，尽管构成共识系统  $P$  的每个进程  $p$  的状态转换函数是确定的 (deterministic)，但是触发状态转换函数的消息接收和处理是异步的，也就是不确定的 (un-deterministic)，因此不可能在有一个缺陷进程的情况下达成整个系统的确定共识状态。

#### 5.1.4 CAP 定理

在 2000 年的分布式计算原则大会 (Symposium on Principles of Distributed Computing 2000) 上，Eric Brewer 做了一个主旨演讲，该演讲分析了当时由于数据量的增长，新出现的一种新的数据库模式 BASE。BASE 的意思是基本可用、软状态、最终一致性 (basically available, soft-state, eventual consistency)，这种新模式是为了解决传统基于 ACID (Atomicity, Consistency, Isolation, Durability) 的数据库在分布式环境中扩展性差而产生的。Brewer 的分析结果中提出了 CAP (Consistency, Availability, Partition Tolerance) 定理的假设。

**CAP 定理：** 一个分布式数据库系统有三个需要的属性：①对网络分隔的容忍 (partition



tolerance); ②一致性 (consistency); ③可用性 (availability)。任意一个数据共享系统, 最多只能在三个属性中选择满足两个属性。

Brewer 当时并没有给出证明, 而正式的证明在 2 年后由 Gilbert 和 Lynch 给出。在证明该定理前, Gilbert 和 Lynch 先对 3 个属性进行了严格的定义。

**定义 1. Consistency :** (数据对象原子性) 在所有的操作中, 必须有一个完整的顺序存在使得每个操作就像在单个实体上完成。在一个分布式多节点环境中, 这意味着所有发生在一个写操作完成之后的读操作都必须返回该写操作写入的值。

**定义 2. Availability :** 每个非故障节点在收到请求后必须有回应。这意味着服务用的任何算法最后必须能终止。

**定义 3. Partition Tolerance :** 可以容忍网络丢失从一个节点发到另一节点任意多的消息。

Gilbert 和 Lynch<sup>[4]</sup>采用反证法来证明 CAP 定理。首先假设所有的条件 (consistency, availability, partition tolerance) 能同时成立。因为任意网络至少两个节点可以分别处在两个非空、不重合的网络节点集合  $\{G_1, G_2\}$  中。一个原子对象  $o$  有一个初始值  $v_0$ 。我们定义  $\alpha_1$  是一个执行步骤, 该步骤把在  $G_1$  中的  $o$  值改成  $v_1$ ,  $v_1 \neq v_0$ 。假设  $\alpha_1$  是唯一一个用户请求。另外, 我们假设  $G_1$  和  $G_2$  中的节点互相没有收到来自对方的消息。因为可用性的要求, 我们知道  $\alpha_1$  是完成的, 也就是说  $o$  现在在  $G_1$  有  $v_1$  的值。类似的,  $\alpha_2$  是一个在  $G_2$  中的单一的读  $o$  的执行步骤, 在  $\alpha_2$  的执行过程中,  $G_1$  和  $G_2$  之间没有消息的传递。由于可用性的要求, 我们知道  $\alpha_2$  也是完成的。如果我们执行  $\alpha_1$  和  $\alpha_2$ ,  $G_2$  只能看到  $\alpha_2$ 。因为  $G_2$  没有收到任何从  $G_1$  来的消息, 也看不到  $\alpha_1$  的请求。那么从  $\alpha_2$  读来的结果还是  $v_0$ 。但由于  $\alpha_2$  读的操作是在  $\alpha_1$  写的操作完成之后开始的, 一致性的要求被违反。这就证明了我们不能同时保证三个属性都满足。

CAP 定理对后来的分布式系统产生了深远的影响。它告诉人们, 在三个属性中必须做取舍, 只能同时满足两个属性。而分布式在网络环境下, 由于分区容错是一个常态, 因此在设计分布式系统的时候, 需要在可用性和一致性上做权衡。AWS 后面推出的分布式数据库 Dynamo, 就是一个舍掉一致性的例子。而谷歌的 GFS, 则是在保证一致性的前提下, 舍弃一些可用性。

CAP 定理和 FLP 定理虽然在某些方面有些相似, 都是研究在分布式系统中的一些不能解决的问题并不直接相关, 也就是说, 两个定理并不是等价关系。它们之间的区别在于: FLP 的假设是网络的异步通信, 是可靠的, 消息虽然会有延迟, 但不会丢失。CAP 中的网络分区则会导致消息丢失。另外 FLP 中的故障节点完全从网络中分隔, 不会响应任何请求; 而 CAP 中的可用性要求系统能响应请求。FLP 是关于分布式系统的共识问题, 这个和 CAP 中的一致性有些相似, 但 CAP 主要讨论关于原子性数据存储的一致性, 以及和可用性的关系。

分布式共识算法的设计需要考虑 FLP 和 CAP 定理的限制, 因为分布式系统都会有对一

致性、可用性、可终止性的要求。重要的是在共识协议的设计上对这些属性的取舍，还有就是假设条件上规避 FLP 不可能达成共识的一些前提条件，例如采用同步或半异步通信方式。

## 5.2 强一致性非拜占庭共识算法

在很多分布式系统场景下，并不需要解决拜占庭将军问题。也就是说，在这些分布式系统的实用场景下，其假设条件不需要考虑拜占庭故障，而只是处理一般的宕机恢复故障。在这种情况下，采用非拜占庭容错共识协议往往效率更高。这种共识协议也应用广泛。在这一部分，我们重点介绍 3 个处理宕机恢复故障的协议：Viewstamped Replication、Paxos 和 Raft 协议。

### 5.2.1 Viewstamped Replicaton

Viewstamped Replication，以下简称 VR，是由 Brian Oki 于 1988 年在其导师 Barbara Liskov 指导下的一篇博士论文中发表的第一个分布式共识算法<sup>[6]</sup>，主要用于在分布式系统状态复制服务。

#### 1. VR 正常流程与视图更换

VR 基于状态机复制技术，在很多节点运行副本（replicas），维护状态并给客户端提供服务。VR 的特点有：

- 1) VR 主要是一个复制协议，但它也能提供共识功能；
- 2) VR 只处理宕机故障，一个节点要么是完全运行良好，要么是停机；
- 3) VR 的运行环境是一个异步网络环境，消息可能丢失，接收顺序可能不能保证；同一消息可能会重复接收。

VR 的副本的总数设成  $2f+1$  个，这是在异步通信网络下存在宕机故障的最少副本配置。VR 保证在不超过  $f$  个副本出故障的情况下整个系统的可靠性和可用性，这可以简单地通过投票中占多数的非故障节点来证明其可靠性和可用性。这个多数组  $f+1$  副本通常称为“法定人数”（quorum）。

VR 的架构如图 5-5 所示。

图中：

- 1) 客户端的代码运行在 VR 代理；
- 2) VR 代理和副本通信，将副本的计算结果返回给客户端；
- 3) VR 代码接受客户端的请求，调用服务代码执行协议；
- 4) 服务代码返回结果给 VR 代码，VR 代码再把结果返回给客户端上的 VR 代理。

VR 共识协议的目的是为了保证在客户端并行发出请求，同时存在着副本故障的情况下，所有非故障副本必须能按同一个顺序来执行。VR 中的副本分为主副本（primary）和从



副本 (secondary)。主副本决定客户请求执行的顺序, 从副本执行主副本选择的执行顺序。如果主副本出故障, VR 允许不同的从副本来替代主副本。VR 系统会发生视图转换, 每个视图有一个副本做主副本的角色。其他副本都会关注主副本, 如果主副本看上去出了故障, 其他副本可以通过一个“改变视图”的步骤来选举新的主副本。

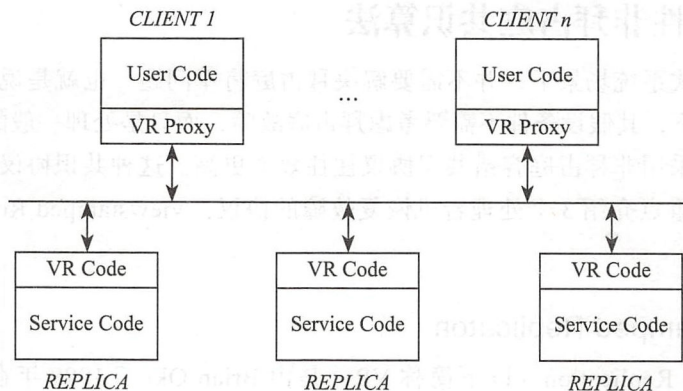


图 5-5 VR 的架构

因此, VR 协议有 3 种场景。第一种是正常情况下处理用户的请求;第二种是视图改变时新主副本的选举;第三种是故障恢复后的副本重新加入。

我们先来看 VR 副本的一些状态参数。VR 的配置 (configuration) 保存所有副本 (2f+1) 的 IP 地址。参数 replicate\_number 保存指向副本的索引, 当前视图号保存在 view\_number 中, 初始值是 0。当前状态保存在 status 中, 取值是 NORMAL、VIEW-CHANGE 和 RECOVERING。op-number 保存当前的请求号, 初始值为 0。日志 log 保存一个存放 op-number 的数组, 它按收到的请求顺序存放。commit\_number 是最近提交的 op-number。client table 是一个保存每个客户最新请求的表, 里面包括给客户的回复。

主副本的视图是通过视图号和配置表计算出来，不需要保存主视图的标志。每个副本按 IP 地址来排序，最小的 IP 地址定为 1，逐渐增加。

在客户端的代理会维护一些状态，包括配置 configuration、当前的视图号以跟踪主副本、客户端的 ID 和客户请求号码。

### (1) 正常情况的 VR 协议

图 5-6 给出了 VR 正常的流程。

具体步骤如下。

1) 客户端发一个请求 (REQUEST)

消息给主副本，让其执行某个操作，  
连同客户端 ID(*client\_id*) 和请求号  
request number。

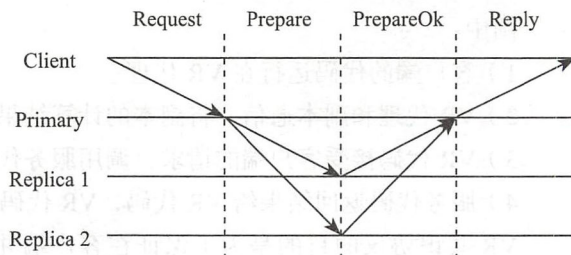


图 5-6 VR 流程

2) 主副本会检查 client table, 如

果 request\_number 小于在 client\_table 表中的请求号,主副本会忽略该请求,并发送回复如果该请求是最近被执行的一个请求。

3) 主副本在 op-number 增加 1,把请求加到日志 log,并用新的请求号更新 client\_table。然后它发一个 PREPARE 消息给其他副本,连同当前的视图号 view\_number、op-number、客户的消息和 commit\_number (最近提交的 op\_number) 一起。

4) 副本按顺序来接收消息,然后把消息加进日志,更新 client\_table 并发一个 PREPAREOK 的消息给主副本。这个消息表示该操作及以前的操作都准备好了。

5) 主副本等收到  $f$  个副本的回复之后才提交操作,把 commit\_number 加 1。当它确认所有在此之前的操作都执行后,会调用服务代码执行当前的操作。一个 REPLY 的消息会发给客户端,包含有 view\_number、request\_number 以及服务代码返回的结果。

6) 其他副本执行请求时,只要保证在此请求之前的请求都被执行,然后执行该请求,在 commit\_number 上增加 1,更新 client\_table,但并不需要直接回复给客户端,只有主副本回复客户端。

7) 如果一个客户端在一定时间内没有收到回复,它将重发请求给所有的副本。如果 VR 已经转到一个新的视图,这个消息将最后到达新的主副本。副本会忽略客户请求,只有主副本处理客户请求。

## (2) 视图转换协议

视图转换是在主副本出现异常的情况下需要更换主副本的协议。该协议也是其他很多共识机制常用的异常处理机制,例如后面的 PBFT 算法的视图更换也是采用相同的协议。

副本监控主副本,它们预期是间断性地能接到主副本消息。一般情况下主副本发“PREPARE”消息,但如果是空闲时段(没有客户请求),它发 COMMIT 消息给副本。如果在一个超时发生而没有收到主副本消息的情况下,副本会执行一个视图转换以选择新的主副本。主副本是轮流选出的。每个副本有一个单一的 IP 地址,下一个主副本是具有最小 IP 地址并正常工作的副本。在 VR 组里的成员都能预计哪一个副本会成为下一个主副本。

视图转换的算法如下。

1) 一个副本注意到需要更换视图,它把 view\_number 增加 1,把状态改成 view\_change,同时发一个 START-VIEW-CHANGE 消息。每个副本根据自身的定时器,或者因为接收到其他副本发送的 START-VIEW-CHANGE 或 DO-VIEW-CHANGE 中高于自身 view-number 的请求来决定是否更换视图。

2) 当一个副本接收到  $f$  个 START-VIEW-CHANGE 消息回应它的 view-number 后,它发 DO-VIEW-CHANGE 给将要成为主副本的节点。该消息包含副本的状态:日志 log、最近的 op\_number、commit\_number、和上次状态是正常状态的视图号。

3) 新的主副本等待接收  $f+1$  个 DO-VIEW-CHANGE 来自其他副本的消息(包括自己)。然后它根据副本的消息将自己的状态更新,把自己的号码作为消息中的 view-number,把状态改成正常(normal)。它发一个 STARTVIEW 消息,连同最新的包括日志(log),commit-





number 和 op-number 给所有的副本节点。

4) 主副本节点可以接收客户端请求。它执行请求并发回给客户端。

5) 当副本收到 STARTVIEW 消息, 它们将根据消息来更新自身的状态。状态更新后, 它们对在日志 (log) 中没有提交的操作, 发 PREPAREOK 消息给新主副本。然后执行这些操作以和新主副本同步。

6) 当主副本接收到  $f$  个 PREPAREOK 消息后, 主副本将提交操作, 并调用服务代码。

## 2. VR 副本节点故障恢复协议

当一个副本从宕机状态恢复时, 在恢复到宕机前的状态之前, 它不能参与处理请求和视图转换。否则系统就会出错。VR 副本需要记住它所执行的操作。VR 副本节点的状态是保存在别的副本节点上, 不需要通过磁盘存储来保持状态。VR 副本的状态可以通过恢复协议来获得。

当一个节点从宕机状态恢复的时候, 它把它的状态置为恢复 (RECOVERING) 同时执行恢复协议。恢复协议步骤如下。

1) 恢复节点发一个 RECOVERY 消息给所有其他副本节点, 连同一个随机数。

2) 收到消息的副本节点, 如果自身是处于正常状态, 它会回复恢复节点一个 RECOVERY-RESPONSE 的消息, 包含它的视图号码和收到的随机数。如果是主副本节点, 它将把自己的日志 (log)、op-number 和 commit-number 一同发回。

3) 当恢复副本收到  $f+1$  个 RECOVERY-RESPONSE 消息, 包括一个从主副本收到的消息, 它将更新自己的状态并把状态设置成正常状态。随机数的作用是保证恢复副本节点只接收对这次恢复的 RECOVERY-RESPONSE 消息, 而不是以前的。

## 3. VR 共识协议小结

VR 是第一个正式发表的共识协议, 它主要用于主从节点状态的复制, 同时也提供一个可行的共识协议。它处理的是在异步通信下存在宕机恢复故障的一致性共识问题。只要在总数不超过一半的故障节点情况下, VR 系统就能够达成共识。需要注意的是, VR 虽然也是一个异步通信环境的共识协议, 但是与 FLP 中的异步通信假设还是不一样。在 FLP 的异步通信假设中, 没有一个同步的时钟可以访问, 也就是说不能依靠超时来判断。也假设无法通过监控来判断进程是否出现故障, 因此有 FLP 不可形成共识的结论。而 VR 则可以依靠超时来判断一个节点是否出故障。因此, VR 可以达成共识与 FLP 的不可能达成共识的结论并不冲突, 只是各自的假设条件不一样。如果严格按照 FLP 的条件来衡量, 那么 VR 可以保证安全性, 但 VR 不能保证活性, 也就是说不能保证在任何情况下, 不出现不断更换视图的情况而无法进入正常共识流程。

## 5.2.2 Paxos 共识算法

与 VR 算法类似, Paxos 共识算法也是一个在异步通信环境的共识算法, 能容忍半数以



下的宕机恢复（非拜占庭故障）节点并达成共识。Paxos 由 Leslie Lamport 于 1998 年发表<sup>[6]</sup>，后面成为强一致性共识算法的代表。大部分后来的强一致共识算法都是 Paxos 的变种。

Paxos 共识系统中有五种角色，每个节点可以同时扮演多种角色。

1) Client (客户端): 客户端向分布式系统发出请求并等待回复，例如更新一个分布式文件系统。

2) Proposer (提议者): 一个 proposer 是接收客户端请求并按其要求提议系统接受特定值的节点。在任何时候都可以有多个提议者。

3) Acceptor (接受者): 一个从提议者那里接收提议的投票节点。接受者可以接受或拒绝提议。

4) Learner (学习者): 一个不参与决策流程，但是必须从提议者和接受者那里了解最后的选择结果，然后采取行动，例如执行请求，返回结果给客户端。

5) Leader (领导者): 在所有提议者 (proposer) 中选出一个领袖，由该领袖来主导提议的发送、投票和最后选择。

Paxos 的提议  $proposal(x, n)$  由提议值  $x$  和一个提议号  $n$  构成。当一个提议者发出一个新的提议，它会选择一个更大的、唯一的提议号。

一个接受者 (acceptor) 看到一个比任何它原有接收到的提议都大的  $n$ ，就会接受该提议。接受者可以接收任意多的提议。接收到的协议不一定会被选择。可以选择多个提议，但这些选择的提议中，提议值必须相同。

Paxos 协议分为两个阶段。

### 1. 准备阶段 (Prepare)

1) 提议者 proposer 发一个  $prepare(x, n)$  消息给 acceptor 的多数集，其中  $n$  必须比提议者以前用过的提议号大。该消息告知所有的接受者希望提议  $propose(x, n)$ 。多数集由提议者选择，保证至少有  $f+1$  个接受者接收（假设接受者至少  $2f+1$ ， $f$  为可能出现的最多故障数）。

2) 接受者 (acceptor) 收到该消息后，如果  $n$  是大于它原来接受到的所有提议号，接受者返回一个过去接受到的提议  $proposal(y, m)$  给提议者， $m$  是过去接收到的最大提议号。同时保证将不接受任何小于  $n$  的提议。如果接受者过去从来没收到提议，则返回  $(\emptyset, 0)$ 。如果  $n$  不是最大的提议号，接受者可以不回答提议者，也可以发送拒绝消息 (Nack) 表示不同意对该提议的共识，否则接受者将把  $n$  作为最大提议号， $x$  作为最后收到的提议值记录下来。

### 2. 提议阶段 (Propose)

1) 如果提议者至少收到  $f+1$  个回复，它将发出正式提议。如果前面只接收到  $acc(\emptyset, 0)$ ，它只发它自己提议的  $x$  值，否则的话将采用接收到的最大提议号  $m$  中的  $y$  值。在发出的提议中，它还是采用自己的提议号  $n$ ，即发出  $propose(y, n)$  的消息给接受者的多数集。

2) 接受者收到提议后，如果它在过去没有向其他提议者保证过只接受大于  $n$  的提议，那该接受者必须接受该提议。接受者把  $y$  值记录下来，并发一个接受  $Ack(y, n)$  消息给提议





者 (proposer) 和学习者 (learner)，表示接受该提议。在表示接受提议时，接受者实际上承认了该提议者为 leader (领袖)。如果接受者在过去曾经向其他提议者保证过只接受大于  $n$  的提议，那么该接受者将忽略这次收到的提议。

3) 提议者在收到  $f+1$  个  $Ack(y, n)$  个回复，则该轮共识达成。否则需要用一个更大的提议号来发起新一轮共识。如果多个提议者发出互相冲突的提议，也会使得共识难以达成。

Paxos 提议的正常流程如图 5-7 所示。

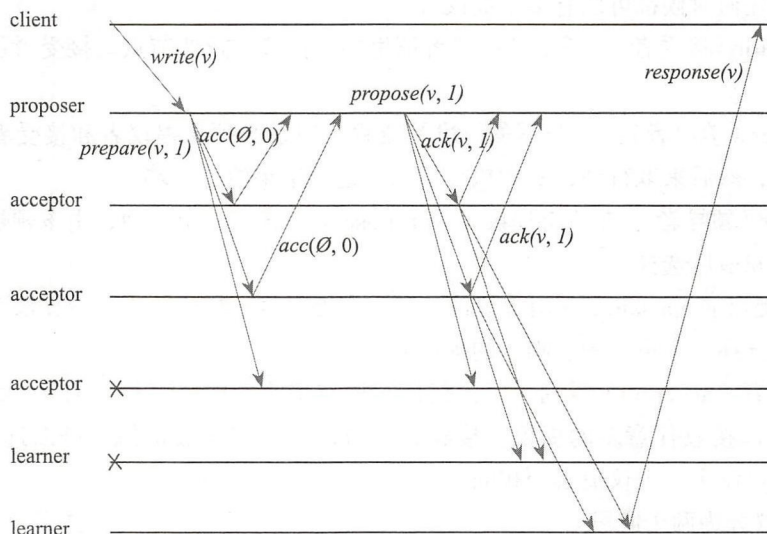


图 5-7 Paxos 正常流程

### 3. Paxos 共识算法小结

Paxos 是强一致、非拜占庭故障的典型共识算法，它在异步环境下能保证在故障节点不超过  $1/2$  的情况下实现共识的安全性，也就是说，能保证如果一个值被选择，其他节点只能选择相同的值。但它却不能保证活性，这意味着在某些特殊情况下，可能不能在有限的时间内达成共识。例如由于网络分隔的原因使得提议者得不到多数接受者的回复的情况。因此，Paxos 和 FLP 并不冲突。

Paxos 协议和 VR 非常类似，不同的地方在于 VR 是一个被动状态机复制协议，它关注的是相同顺序的操作指令被各副本按顺序执行，VR 是由主节点主导，只有主节点有决定性 (deterministic) 状态机，从节点“被动”听从主节点指令的复制协议；而 Paxos 则更多是一个“主动”状态机复制共识协议，各节点自带决定性 (deterministic) 状态机，各节点同意同一个值或同一个执行顺序。

### 5.2.3 其他类 Paxos 共识协议

Paxos 共识协议是一个典型的高效非拜占庭故障的共识协议，但 Paxos 的缺点是协议比



较复杂, 比较难以理解, 特别是在实施上会碰到不少问题。因此后面很多共识协议的设计是基于 Paxos, 但目的是设计适合于不同场景、比较容易理解、容易实现的共识协议。例如像 Raft、Chubby、Zookeeper、etcd 等都是应用比较广泛的、基于 Paxos 的共识协议。下面我们重点介绍一下 Raft, 它在联盟链和私有链上都得到了比较广泛的应用。

### 1. Raft 共识协议

Raft 共识协议是近来比较流行的一个基于 Paxos 的共识协议。Raft 最初是一个用于管理复制日志的共识算法<sup>[7]</sup>, 也是基于状态机复制的协议。它是一个为真实世界应用建立的协议, 主要注重协议的落地性和可理解性。Raft 是在非拜占庭故障下达成共识的强一致协议。

在区块链系统中, 使用 Raft 实现记账共识的过程可以描述如下: 首先选举一个首领 leader, 接着赋予 leader 完全的权力管理记账。leader 从客户端接收记账请求, 完成记账操作, 生成区块, 并复制到其他记账节点。leader 简化了记账操作的管理。例如, leader 能够决定是否接受新的交易记录项而无须考虑其他的记账节点, leader 可能失效或与其他节点失去联系, 这时, 系统就会选出新的 leader。

给定 leader 方法, Raft 将共识问题分解为三个相对独立的子问题。

❑ leader 选举: 现有的 leader 失效时, 必须选出新 leader。

❑ 记账: leader 必须接受来自客户端的交易记录项, 在参与共识记账的节点中进行复制, 并使其他的记账节点认可交易所对应的区块。

❑ 安全: 若某个记账节点对其状态机应用了某个特定的区块项, 其他的服务器不能对同一个区块索引应用不同的命令。

#### (1) Raft 基础

一个 Raft 集群通常包含 5 个服务器, 允许系统有 2 个故障服务器。每个服务器处于 3 个状态之一: leader、follower 或 candidate。正常操作状态下, 仅有一个 leader, 其他的服务器均为 follower。follower 是被动的, 不会对自身发出请求而是对来自 leader 与 candidate 的请求做出响应。leader 处理所有的客户端请求 (若客户端联系 follower, 则该 follower 将转发给 leader)。candidate 状态用来选举 leader。

Raft 阶段主要分为两个, 首先是 leader 选举过程, 然后在选举出来的 leader 基础上进行正常操作, 如日志复制、记账等。

#### (2) leader 选举

若 follower 在选举超时时间内未收到 leader 的心跳消息, 则转换为 candidate 状态。为了避免选举冲突, 这个超时时间是一个 150 ~ 300ms 之间的随机数。

一般而言, 在 Raft 系统中:

1) 任何一个服务器都可以成为一个候选者 candidate, 它向其他服务器 follower 发出要求选举自己的请求。

2) 其他服务器同意了, 发出 OK。如果在这个过程中, 有一个 follower 宕机, 没有收





到请求选举的要求，此时候选者可以自己选自己，只要达到  $N/2+1$  的大多数票（ $N$  是所有节点的总数），候选人还是可以成为 leader 的。

3) 这样这个候选者就成为 leader 领导人，它可以向选民，也就是 follower 们发出指令，比如进行记账。

4) 以后通过心跳进行记账的通知。

5) 如果一旦这个 leader 宕机了，那么 follower 中有一个成为候选者，发出邀票选举。

6) follower 同意后，其成为 leader，继续承担记账等指导工作。

### (3) 记账过程

Raft 的记账过程按以下步骤完成。

1) 假设 leader 领导人已经选出，这时客户端发出增加一个交易记录的记账要求；

2) leader 要求 follower 遵从它的指令，都将这个新的交易记录追加到他们各自的账本中；

3) 大多数 follower 服务器将交易记录写入账本后，确认追加成功，发出确认成功信息；

4) 在下一个心跳中，leader 会通知所有 follower 更新确认的项目。

对于每个新的交易记录，重复上述过程。

如果在这一过程中发生了网络通信故障，使得 leader 不能访问大多数 follower 了，那么 leader 只能正常更新它能访问的那些 follower 服务器。而大多数的服务器 follower 因为有了 leader，他们重新选举一个候选者作为 leader，然后这个 leader 作为代表与外界打交道，如果外界要求其添加新的交易记录，这个新的 leader 就按上述步骤通知大多数 follower。如果这时网络故障修复了，那么原先的 leader 就变成 follower，在失联阶段这个老 leader 的任何更新都不能算确认，都要回滚，接受新的 leader 的新的更新。

## 2. 其他协议

其他基于 Paxos 的共识算法有 Chubby、ZooKeeper、etcd 等。这些共识算法广泛用于各种分布式系统。例如谷歌的 GFS 和 BigTable 使用 Chubby 的分布式锁协议；Yahoo 的 Hadoop 系统和 OpenStack、Mesos 等采用了 ZooKeeper；Kubernetes 和 CoreOS 采用了 etcd 协议等。除了超级账本的 Fabric1.0 用了基于 ZooKeeper 的 Kafka 做排序引擎外，这些共识协议用在区块链的场景目前不是很多见，因此我们这里不做详细叙述。

### 5.2.4 强一致性非拜占庭共识算法小结

共识问题是分布式系统中一个最重要的问题。在很多场景，包括服务器副本复制、日志复制、同步服务、分布式锁、领袖选举、元数据管理、消息队列中都会有共识的需求。分布式系统之所以能协同工作，归根结底有赖于共识协议的协调。拜占庭将军问题是一个形象的比喻，说明在一个同步通信系统中，当有超过  $1/3$  系统节点总数的节点出现拜占庭故



障时,该分布式系统将不可能达成共识。而 FLP 定理告诉我们,在异步通信情况下,即使有一个非拜占庭故障(如宕机、宕机恢复故障)的存在,就不能保证系统能在有限的时间内达成共识,也就是说,不能同时满足安全性(safety)和活性(liveness)的要求。分布式系统中著名的 CAP 定理也从另一个角度说明,分布式系统不能同时满足一致性、可用性和网络分区容错性,只能在三者中取其二。

因此,在设计共识算法的时候,要根据实际情况来规避以上的限制。大部分的分布式共识算法,都是在保证安全性,也就是一致性和正确性的前提下,牺牲活性。

Viewstamped Replication 是第一个公开发表的共识协议。后面的共识协议大多基于 Paxos。Paxos 是一个高效而复杂的协议,比较难以理解,也难以实现。因此后来出现了各种不同的实现和变种。例如谷歌的 GFS、Bigtable 就采用了基于 Paxos 的 Chubby 的分布式锁协议;Yahoo 的 Hadoop 系统采用了类似 Paxos 协议的 Zookeeper 协议。Raft 也是为了避免 Paxos 的复杂性而专门设计成易于理解的一个分布式一致性算法。大部分强一致性的非拜占庭故障共识协议都能容忍少于系统总节点  $1/2$  的宕机恢复故障。

在私有链和联盟链的场景下,通常共识算法有强一致性要求,同时对共识效率要求高。另外一般安全性要比公有链场景高,一般来说不会经常存在拜占庭故障。因此,在一些场景下,可以考虑采用非拜占庭容错的分布式共识算法。在 Hyperledger 的 Fabric 项目中,未来共识模块被设计成可插拔的模块,支持像 PBFT、Raft 等共识算法。目前在区块链项目中比较常见的非拜占庭容错共识算法是 Raft。

## 5.3 强一致性拜占庭容错共识算法

FLP 定理给出了在异步通信系统对分布式共识的理论限制,但由于同步通信的可扩展性小,大部分分布式系统不能依靠同步通信,否则性能和效率将非常低。因此寻找一种实用的在异步通信环境下解决拜占庭将军问题的算法一直是分布式计算领域中的一个重要问题。

在区块链网络中,由于应用场景的不同、所设计的目标各异,不同的区块链系统采用了不同的共识算法。一般来说,私有链和联盟链情况对一致性、正确性有很强要求,一般来说要采用强一致性的共识算法。而公有链情况对一致性和正确性无法做到百分之百,通常采用最终一致性(Eventual Consistency)的共识算法。

下面我们先来介绍适合私有链和联盟链场景的拜占庭容错系统。

区块链网络的记账共识和拜占庭将军问题是相似的。参与共识记账的每一个记账节点相当于将军,节点之间的消息传递相当于信使,某些节点可能由于各种原因而产生错误的信息并传达给其他节点。通常,这些发生故障的节点被称为拜占庭节点,而正常的节点即为非拜占庭节点。

拜占庭容错系统是一个拥有  $n$  台节点的系统,整个系统对于每一个请求满足以下的条件:





- 1) 所有非拜占庭节点使用相同的输入信息, 产生同样的结果;
- 2) 如果输入的信息正确, 那么所有非拜占庭节点必须接受这个信息, 并计算相应的结果。

与此同时, 在拜占庭系统的实际运行过程中, 还需要假设整个系统中拜占庭节点不超过  $m$  台, 并且每个请求还需要满足两个指标。

- 安全性: 任何已经完成的请求都不会被更改, 它可以被以后的请求看到;
- 活性: 可以接受并且执行非拜占庭客户端的请求, 不会被任何因素影响而导致非拜占庭客户端的请求不能执行。

拜占庭系统普遍采用的假设条件包括:

- 1) 拜占庭节点的行为可以是任意的, 拜占庭节点之间可以共谋;
- 2) 节点之间的错误是不相关的;
- 3) 节点之间通过异步网络连接, 网络中的消息可能丢失、乱序并延时到达, 但大部分协议假设消息在有限的时间里能传递到目的地 (该假设是避免 FLP 的不可能共识条件);
- 4) 第三方可以探测到服务器之间传递的信息, 但是不能篡改、伪造信息的内容和验证信息的完整性。

### 1. 实用的拜占庭容错系统

原始的拜占庭容错系统<sup>[7]</sup>由于需要展示其理论上的可行性而缺乏实用性, 另外需要额外的时钟同步机制支持, 算法的复杂度也随节点增加而指数级增加。Castro 和 Liskov 在 1999 年提出实用拜占庭容错系统 (Practical Byzantine Fault Tolerance, PBFT)<sup>[8]</sup>, 降低了拜占庭协议的运行复杂度, 从指数级别降低到多项式级别 (Polynomial), 使拜占庭协议在分布式系统中的应用成为可能。

PBFT 是一类状态机拜占庭系统<sup>[9]</sup>, 要求整个系统共同维护一个状态, 所有节点采取的行动一致。为此, 需要运行三类基本协议, 包括一致性协议、检查点协议和视图更换协议。我们主要关注支持系统日常运行的一致性协议。

一致性协议要求来自客户端的请求在每个服务节点上都按照一个确定的顺序执行。这个协议把服务器节点分为两类: 主节点和从节点, 其中主节点仅一个。在协议中, 主节点负责将客户端的请求排序, 从节点按照主节点提供的顺序执行请求。每个服务器节点在同样的配置信息下工作, 该配置信息被称为视图。主节点更换, 视图也随之变化。

一致性协议至少包含 5 个阶段: 请求 (request)、序号分配 (pre-prepare)、相互交互 (prepare)、序号确认 (commit) 和响应 (reply)。

PBFT 的一致性协议正常流程如图 5-8 所示。PBFT 系统通常假定故障节点数为  $f$  个, 而整个服务节点数为  $3f+1$  个。每一个客户端的请求需要经过 5 个阶段, 通过采用两次两两交互的方式在服务器达成一致之后再执行客户端的请求。由于客户端不能从服务器端获得任何服务器运行状态的信息, PBFT 中主节点是否发生错误只能由服务器监测。如果服务器在一段时间内都不能完成客户端的请求, 则会触发视图更换协议。视图更换协议与 VR 的



视图更换协议一样，请参考 5.2.1 节。

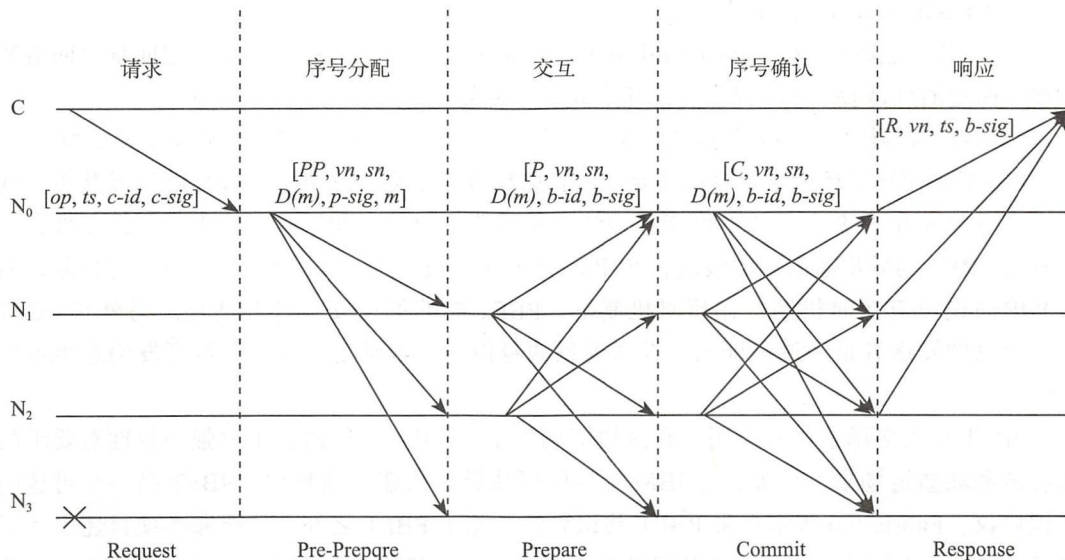


图 5-8 PBFT 协议正常流程

图 5-8 显示的是一个简化的 PBFT 的协议通信模式，其中 C 为客户端， $N_0 \sim N_3$  表示服务节点， $N_0$  为主节点，“×”表示  $N_3$  为故障节点。虚线是各阶段边界，箭头表示消息从消息源节点发送到接收节点。整个协议的基本过程如下。

### (1) Request 请求阶段

客户端发送请求给主节点，请求消息  $m=[op, ts, c-id, c-sig]$ ，其中包含需要执行的操作  $op$ ，时间戳  $ts$ ， $cs-id$  是客户端 ID，以及客户端签名  $c-sig$ 。时间戳是为了保证命令只被执行一次，客户端的签名是为了客户认证和权限控制。

### (2) Pre-Prepare (序号分配) 阶段

当主节点接收请求后，主节点给请求赋值一个序列号  $sn$ ，广播序号分配消息和客户端的请求消息  $m$ ，并将构造 Pre-Prepare 消息  $[PP, vn, sn, D(m), p-sig, m]$  给各从节点，其中  $PP$  表示 Pre-Prepare 消息、 $vn$  是视图号、 $D(m)$  是客户消息的摘要（哈希值）、 $p-sig$  是主节点的签名、 $m$  是客户消息。序列号是为了保证命令执行的顺序，视图号让从节点记录当前视图。主节点签名是为了让从节点认证主节点身份。消息摘要保证消息没有被篡改。

### (3) Prepare (交互) 阶段

从节点接收 Pre-Prepare 消息，向其他服务节点广播 Prepare 消息  $[P, vn, sn, D(m), b-id, b-sig]$ ，其中  $P$  表示 Prepare 消息、 $b-id$  是从节点 id、 $b-sig$  是从节点签名。

### (4) Commit (序号确认) 阶段

从节点在收到  $2f+1$  个 Prepare 消息后，对视图内的请求和次序进行验证，广播 Commit 消息  $[C, vn, sn, D(m), b-id, b-sig]$ ，其中  $C$  表示 Commit 消息。执行收到的客户端请求并给





客户端以响应。

### (5) Response (响应) 阶段

1) 当各节点收到  $2f+1$  个 Commit 消息后, 将执行操作, 并提交。同时把回复返回给客户端。回复消息是  $[R, vn, ts, b-sig]$ , 其中  $R$  表示回复消息。

2) 客户端等待来自不同节点的响应, 若有  $f+1$  个响应相同, 则该响应即为运算的结果。

PBFT 能在异步环境下, 容忍不足  $1/3$  的总数节点是拜占庭节点, 并能够达成共识。这个表面看起来和 FLP 定理的不可能共识结论相违背, 但其实如果我们检查其假设条件, 可以看到 PBFT 的异步通信环境假设和 FLP 的异步通信假设不一样。PBFT 中有超时机制, 而 FLP 中假设没有超时机制。在超时机制下, PBFT 能够保证安全性和活性。另外在 PBFT 中, 消息的发送者是经过认证的, 发送消息的身份可以被确定, 节点有没有发消息也可以验证。

PBFT 在很多场景都有应用, 在区块链场景中, PBFT 一般适合于对强一致性有要求的私有链和联盟链场景。例如, 在 IBM 主导的区块链超级账本项目中, PBFT 是一个可选的共识协议。Fabric 0.6 版本自带 PBFT 共识算法。除了 PBFT 之外, 超级账本项目还引入了基于 PBFT 的自用共识协议。它的目的是希望在 PBFT 基础之上能够对节点的输出也做好共识, 这是因为超级账本项目的一个重要功能是提供区块链之上的智能合约, 即在区块链上执行的一段代码, 因此它会带来区块链账本上最终状态的不确定, 为此这个自有共识协议会在 PBFT 实现的基础上, 引入代码执行结果签名进行验证。

## 2. 小结

本节介绍了强一致性拜占庭容错共识算法中最为典型的实用拜占庭容错 (PBFT) 算法。在区块链的企业级应用, 例如大部分联盟链或私有链, 通常需要保证系统各节点的强一致性, 也就是说, 保证系统各节点按相同的顺序执行相同的命令, 形成相同的结果。这些结果一旦形成共识, 就不可改变。特别有拜占庭故障的环境, 需要有强一致的拜占庭容错共识算法, 而 PBFT 是这一类共识算法的基础。其他的拜占庭容错共识算法还有 Fab Paxos、XFS、Zyzyva、SBFT 等。

## 5.4 非强一致共识算法 PoW 机制

比特币系统<sup>[9]</sup>的重要概念是一个基于互联网的去中心化分布式账本, 该分布式账本以区块链形式保存, 每个区块相当于账本页, 区块中记录的信息主体即为相应的交易内容。传统账本为了保证交易内容的唯一性, 一般要求记账行为是中心化的行为。然而, 中心化所引发的单点失败可能导致整个系统面临危机甚至崩溃。另外, 中心化的记账要求所有交易相关方信任中心化记账机构。这种信任很多时候需要很高的成本才能维持, 例如银行等金融机构需要有雄厚的资本来维持信任。

去中心化记账可以克服中心化记账成本高、安全脆弱的缺点, 但需要解决记账行为的



一致性问题。从去中心化账本系统的角度，每个加入这个系统的节点都要保存一份完整的账本，但每个节点却不能同时记账。因为节点处于不同的环境，接收到不同的信息，如果同时记账的话，必然会导致账本的不一致，造成混乱。因此，需要达成哪个节点有权记账的共识。比特币区块链通过基于算力的随机性竞争记账的方式选出一个节点，来记录一个区块的交易信息，向其他节点同步这个新增的区块信息，解决去中心化的记账系统的一致性问题。

比特币系统设计了以每个节点的计算能力，即“算力”来竞争记账权的机制。在比特币系统中，大约每 10 分钟进行一轮算力竞赛，竞赛的胜利者就获得一次记账的权力，并向其他节点同步新增账本信息。

然而，在一个去中心化的系统中，谁有权判定竞争的结果呢？比特币系统是通过 PoW 机制完成的。

PoW 系统的主要特征是计算的不对称性。工作端需要做一定难度的工作得出一个结果，验证方却很容易通过结果来检查出客户端是不是做了相应的工作。

举个例子，对于给定字符串 "blockchain"，我们给出的工作量要求是，可以在这个字符串后面连接一个称为 nonce 的整数值串，对连接后的字符串进行 SHA-256 哈希运算，如果得到的哈希结果（以十六进制的形式表示）是以若干个 0 开头的，则验证通过。为了达到这个工作量证明的目标，我们需要不停地枚举 nonce 值，然后对得到的新字符串进行 SHA-256 哈希运算。按照这个规则，我们需要经过 2688 次计算才能找到前 3 位均为 0 的哈希值；而要找到前 6 位均为 0 的哈希值，则需进行 620 969 次计算。

```
blockchain1 → 4bfb943cba9fb9926df93f33c17d64b378d56714e8a29c6ba8bdc9690cea8e27
blockchain2 → 01181212a283e760929f6b1628d903127c65e6fb5a9ad7fe94b790e699269221
.....
blockchain515 → 0074448bea8027bebd6333d3aa12fd11641e051911c5bab661a9b849b83958a7
.....
blockchain2688 → 0009b257eb8cf9eba179ab2be74d446fa1c59f0adfa8814260f52ae0016dd50f
.....
blockchain48851: 00000b3d96b4db1a976d3a69829aabef8bafa35ab5871e084211a16d3a4f385c
.....
blockchain6200969: 000000db7fa334aef754b51792cff6c880cd286c5f490d5cf73f658d9576d424
```

对于特定字符串后接随机 nonce 值所构成的串来说，要找到一个 nonce 值，满足前  $n$  位均为 0 的 SHA-256 值，需要多次进行哈希值的计算。一般来说， $n$  值越大，需要完成的哈希计算量也越大。由于哈希值的伪随机特性，要寻求 4 个前导 0 的哈希值，预期大概要进行  $2^{16}$  次尝试，这个数学期望的计算次数，就是所要求的“工作量”。

比特币网络中任何一个节点，如果想生成一个新的区块并写入区块链，必须解出比特币网络出的 PoW 问题。这道题关键的三个要素是工作量证明函数、区块及难度值。工作量证明函数是这道题的计算方法，区块决定了这道题的输入数据，难度值决定了这道题所需要的计算量。





## 1. 工作量证明函数

比特币系统中使用的工作量证明函数正是 SHA-256。

SHA 是一个密码哈希函数家族，主要适用于数字签名标准。SHA-256 就是这个函数家族中的一个，是输出值为 256 位的哈希算法。到目前为止，还没有出现对 SHA-256 算法的有效攻击。具体见第 4 章的讲解。

## 2. 区块

比特币的区块由区块头及该区块所包含的交易列表组成。区块头的大小为 80 字节，由 4 字节的版本号、32 字节的上一个区块头的哈希值、32 字节的 Merkle 根哈希值、4 字节的时间戳（当前时间）、4 字节的当前难度值、4 字节的随机数组成。区块包含的交易列表，也叫区块体，附加在区块头后面，其中的第一笔交易是 coinbase 交易，这是一笔为了让矿工获得奖励及手续费的特殊交易。

区块的大致结构如图 5-9 所示。

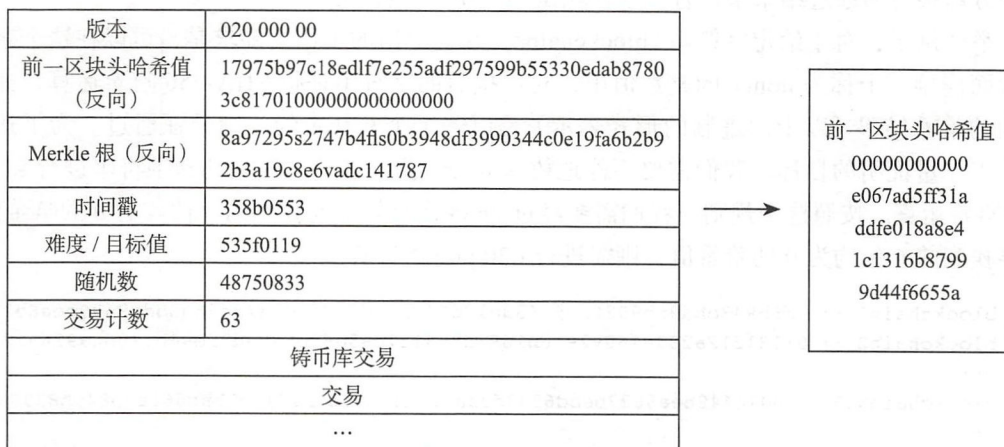


图 5-9 区块的结构

拥有 80 字节固定长度的区块头，就是用于比特币工作量证明的输入字符串。因此，为了使区块头能体现区块所包含的所有交易，在区块的构造过程中，需要将该区块要包含的交易列表，通过 Merkle 树算法生成 Merkle 根哈希值，并以此作为交易列表的哈希值存到区块头中。其中 Merkle 树的算法图解如图 5-10 所示。

图 5-10 展示了一个具有 4 个交易记录的 Merkle 树的根哈希值的计算过程。首先以这 4 个交易作为叶子节点构造一棵完全二叉树，然后通过哈希值的计算，将这棵二叉树转化为 Merkle 树。

首先对 4 个交易记录  $T_{xa} \sim T_{xd}$ ，分别计算各自的哈希值  $H_a \sim H_d$ ，然后计算两个中间节点的哈希值  $H_{AB} = \text{Hash}(H_a + H_b)$  和  $H_{CD} = \text{Hash}(H_c + H_d)$ ，最后计算出根节点的哈希值  $H_{ABCD} = \text{Hash}(H_{AB} + H_{CD})$ 。

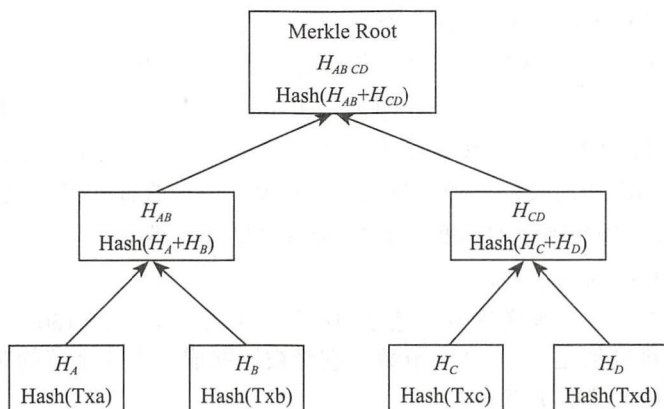


图 5-10 带 4 个交易记录的 Merkle 树根哈希值的计算

构造出来的区块链如图 5-11 所示。

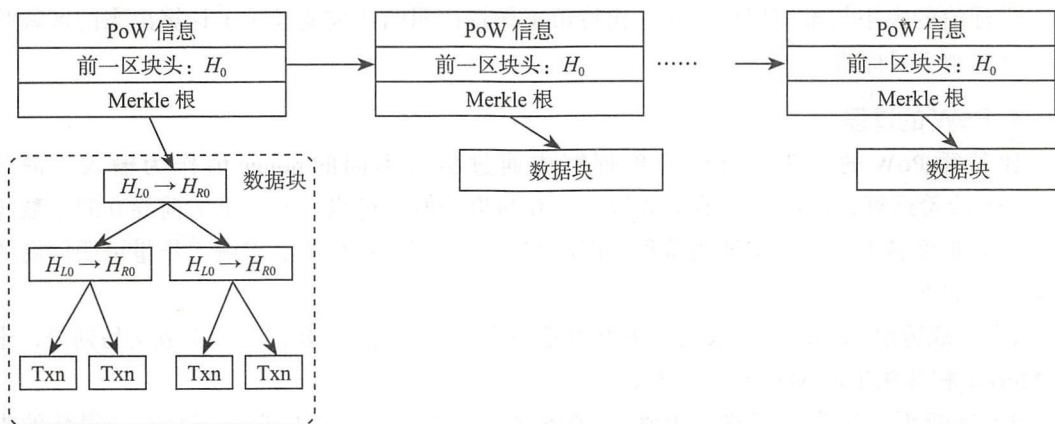


图 5-11 区块链的简化结构

由图 5-7 所示的简化的区块链结构我们可以发现, 所有在给定时间范围需要记录的交易信息被构造成一个 Merkle 树, 区块头中包含了指向这个 Merkle 树的哈希指针, 关联了与该区块相关的交易数据。另一方面, 该区块中也包含了指向前一区块的哈希指针, 使得记录了不同交易的单个区块被关联起来, 形成区块链。

### 3. 难度值

难度值是比特币系统中的节点在生成区块时的重要参考指标, 它决定了节点大约需要经过多少次哈希运算才能产生一个合法的区块。比特币的区块大约每 10 分钟生成一个, 如果要在不同的全网算力条件下, 新区块的产生都基本保持这个速率, 难度值就必须根据全网算力的变化进行调整。简单地说, 难度值根据全网节点计算能力变化来设定, 使得新区



块产生速率能都保持在大约每 10 分钟一个。

难度的调整是在每个完整节点中独立自动发生的。每 2 016 个区块，所有节点都会按统一的公式自动调整难度，这个公式是由上个 2 016 个区块的花费时长与期望时长（期望时长为 20 160 分钟，即两周，是按每 10 分钟一个区块的产生速率计算出的总时长）比较得出的，根据实际时长与期望时长的比值，进行相应调整（或变难或变易）。也就是说，如果区块产生的速率比 10 分钟快则增加难度，比 10 分钟慢则降低难度。

这个公式可以总结为如下形式：

新难度值 = 旧难度值 \* (过去 2 016 个区块花费时长 / 20 160 分钟)

在比特币中，难度值是一个 32 字节数，它将被压缩成一个 4 字节的难度位存在区块头中。目标值和难度位的换算关系是：

目标值 = 难度位后三字节 \* (2<sup>8\*(难度位第一字节-3)</sup>)

目标值 = 最大目标值 / 难度值

其中最大目标值为一个恒定值：

0x00000000FF

目标值的大小与难度值成反比。比特币工作量证明的达成就是矿工计算出来的区块哈希值必须小于目标值。

#### 4. PoW 的过程

比特币 PoW 的过程，可以简单理解成通过尝试不同的 nonce 值作为输入，进行 SHA256 哈希运算，找出满足给定数量前导 0 的哈希值的过程。而要求的前导 0 的个数越多，代表难度越大。工作量证明流程见图 5-12。可以把比特币节点求解工作量证明问题的步骤归纳如下。

1) 生成铸币 (coinbase) 交易，并与其他所有准备打包进区块的交易组成交易列表，通过 Merkle 树算法生成 Merkle 根哈希；

2) 判断距上次难度调整是否到达 2016 区块，如果是，进行难度调整，获得新的难度位；

3) 通过难度位计算目标值；

4) 把版本号、前一区块头哈希值、Merkle 根哈希值、时间戳、难度位和随机数字段组装成区块头，将区块头的 80 字节数据作为工作量证明的输入；

5) 不停地变更区块头中的随机数，即 nonce 的数值，并对每次变更后的区块头做双重 SHA-256 运算（即 SHA-256(SHA-256(Block\_Header))），将结果值与当前网络的目标值做对比，如果小于目标值，则解题成功，工作量证明完成。

比特币的工作量证明，就是俗称“挖矿”所做的主要工作。理解工作量证明机制，将为我们进一步理解比特币区块链的共识机制奠定基础。

#### 5. 基于 PoW 的共识记账

我们以比特币网络的共识记账为例，来说明基于 PoW 的共识记账过程。

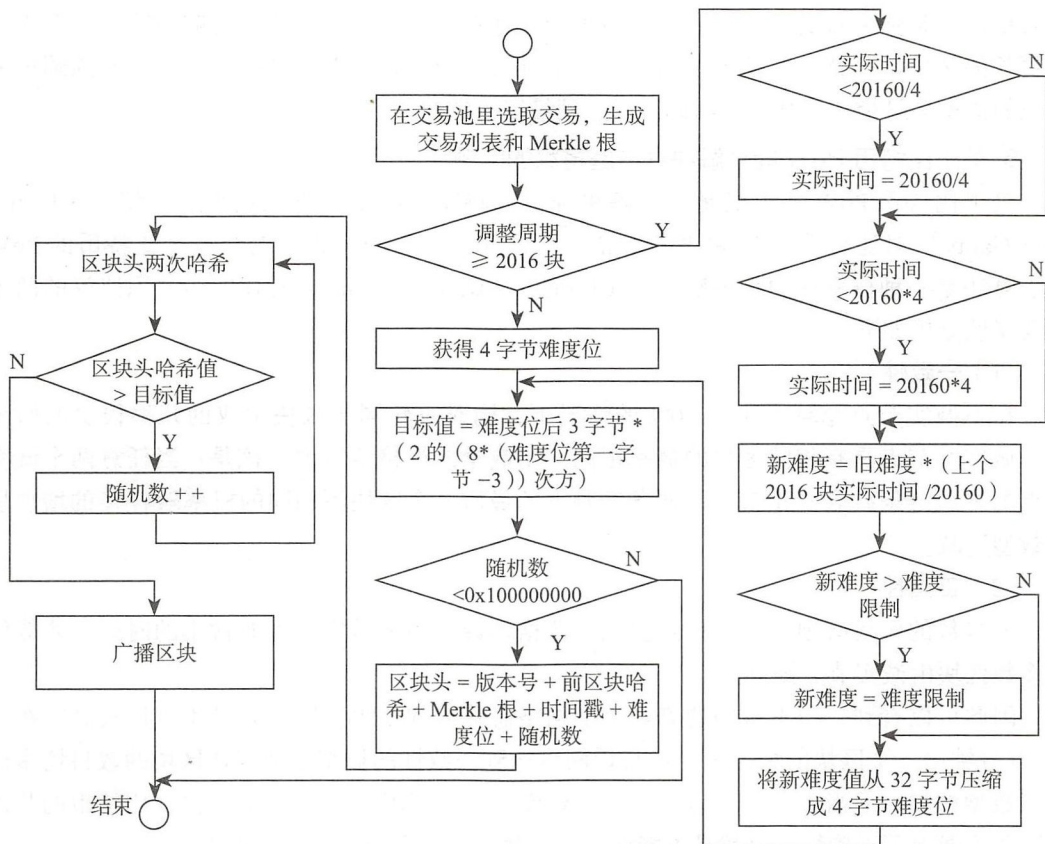


图 5-12 比特币 PoW 共识算法流程

基本过程如下。

- 1) 客户端产生新的交易，向全网进行广播要求对交易进行记账；
- 2) 每一个记账节点一旦收到这个请求，将收到的交易信息纳入一个区块中；
- 3) 每个节点都通过 PoW 过程，尝试在自己的区块中找到一个具有足够难度的工作量证明；
- 4) 当某个节点找到了一个工作量证明，它就向全网广播符合难度的区块；
- 5) 当且仅当包含在该区块中的所有交易都是有效的且之前未存在过的，其他节点才认同该区块的有效性；
- 6) 其他节点表示接受该区块，而表示接受的方法则是在跟随该区块的末尾制造新的区块以延长该链条，而将被接受区块的哈希值视为下一新区块的前项哈希值。
- 7) 比特币系统默认最长的区块链为共识区块链，被最长区块链包含的区块就是系统对区内交易达成的共识的交易。

通过上述的记账过程，客户端所要求记录的交易信息被写入了各个记账节点的区块链中，形成一个分布式的、高概率的一致性账本。注意比特币的工作量证明机制是最终一致



性共识而不是强一致的共识算法。因为理论上存在一个概率，只要有足够的算力，形成一个更长的区块链，就可以把过去的共识结果推翻。当然，随着区块链的延长，要推翻过去共识的概率会呈指数型减少，因此可以达到最终一致性。

## 6. 关于比特币 PoW 能否解决拜占庭将军的问题

关于比特币 PoW 共识机制能否解决拜占庭将军的问题一直在业界有争议。2015 年，Juan Garay<sup>[10]</sup> 对比特币的 PoW 共识算法进行了正式的分析，得出的结论是比特币的 PoW 共识算法是一种概率性的拜占庭协议（Probabilistic BA）。Garay 对比特币共识协议的两个重要属性分析如下。

### （1）一致性

在不诚实节点总算力小于 50% 的情况下，同时每轮同步区块生成的几率很少的情况下，诚实的节点具有相同区块的概率很高。用数学的严格语言说应该是：当任意两个诚实节点的本地链条截取  $K$  个节点，两条链剩下的最后一个区块不相同的概率随着  $K$  的增加呈指数型递减。

### （2）正确性

大多数的区块必须由诚实节点提供。严格地说，当不诚实算力非常小的时候，才能使大多数区块由诚实节点提供。

因此可以看到，当不诚实的算力小于网络总算力的 50%，同时挖矿难度比较高，在大约 10 分钟出一个区块的情况下，比特币网络达到一致性的概率会随确认区块的数目增多而呈指数型增加。但当不诚实算力具有一定规模，甚至不用接近 50% 的时候，比特币的共识算法并不能保证正确性，也就是不能保证大多数的区块由诚实节点来提供。

因此我们可以看到，比特币的共识算法不适合于私有链和联盟链。首先因为它是一个最终一致性共识算法，而不是一个强一致性共识算法。企业应用需要有强一致的共识算法来保证交易的正确性，而不能依赖概率来决定。第二个原因是其共识效率低。提升共识效率又会牺牲共识协议的安全性。强一致的非拜占庭容错共识算法也不适合于像比特币这样的公有链环境，因为在公有链上一定存在拜占庭节点。而像 PBFT 那样的强一致的拜占庭容错共识算法也由于扩展性有限，不能支持成千上万个记账节点。

另外，比特币通过巧妙的矿工奖励机制来提升网络的安全性。矿工挖矿获得比特币奖励以及记账所得的交易费用使得矿工更希望维护网络的正常运行，而任何破坏网络的非诚信行为都会损害矿工自身的利益。因此，即使有些比特币矿池具备强大的算力，它们都没有作恶的动机，反而有动力维护比特币的正常运行，因为这和它们的切身利益相关。

## 5.5 PoS 机制

PoW 背后的基本概念很简单：工作端提交难于计算但易于验证的已知计算结果，而其他人任何人都能够通过验证这个答案就确信工作端为了求得结果已经完成了相当大量的计算

工作。然而 PoW 机制存在明显的弊端。一方面, PoW 的前提是, 节点和算力是均匀分布的。因为通过 CPU 的计算能力来进行投票, 拥有钱包(节点)数和算力值应该是大致匹配的, 然而随着人们将 CPU 挖矿逐渐升级到 GPU、FPGA 甚至 ASIC 矿机挖矿, 节点数和算力值也渐渐失配。比特币网络每秒可完成数百万亿次 SHA-256 计算, 但这些计算除了使恶意攻击者不能轻易地伪装成几百万个节点和打垮比特币网络, 并没有更多实际的或科学的价值。由于 PoW 非常耗能, 因此比特币的 PoW 也因为巨大的能耗浪费被很多人诟病。当然, 如果从另一方面来看, 相对于允许世界上任何一个人在瞬间就能通过去中心化和半匿名的全球货币网络, 给其他人几乎没有手续费地转账所带来的巨大好处, 它的浪费也许只算是很小的代价。

有鉴于此, 人们提出了一些工作量证明的替代者。PoS 就是其中的一种方法。

### 5.5.1 点点币 PoS 机制

权益证明要求用户证明拥有某些数量的货币(即对货币的权益), 点点币(Peercoin)<sup>[11]</sup>是首先采用权益证明的虚拟货币。它采用一种混合共识机制, 既用 PoW 工作量证明挖矿机制, 同时又用权益证明 PoS 机制。点点币中有两种块, 一种是像比特币挖矿工作量证明产生的区块(区块中含有 Coinbase 交易), 另一种是权益证明产生的区块(区块中含有 Coinstake 区块)。

点点币在 SHA-256 的哈希运算难度方面引入了币龄的概念, 使得工作量难度与交易输入的币龄成反比。在点点币中, 币龄被定义为币的数量与币所拥有的天数的乘积, 这使得币龄能够反映交易时刻用户所拥有的货币数量。

实际上, 点点币的权益证明机制结合了随机化与币龄的概念, 未使用至少 30 天的币可以参与竞争下一区块, 越久和越大的币集有更大的可能去签名下一区块。然而, 一旦币的权益被用于签名一个区块, 则币龄将清为零, 这样必须等待至少 30 天才能签署另一区块。同时, 为防止非常老或非常大的权益控制区块链, 寻找下一区块的最大概率在 90 天后达到最大值, 这一过程保护了网络, 并随着时间逐渐生成新的币而无须消耗大量的计算能力。点点币的开发者声称这将使得恶意攻击变得困难, 因为没有中心化的挖矿池需求, 而且购买半数以上的币的开销似乎超过获得 51% 的工作量证明的哈希计算能力, 在一定程度上避免比特币的 51% 攻击。另外攻击者的币龄会在攻击中受到消耗, 所以对攻击者持续攻击的能力有限制。

在点点币的设计中, 点点币的权益证明产生币的比率是每个币年消耗产生一分币, 以保证未来的低通胀。为了避免出块率的陡然提升, 工作量证明机制的目标值和权益证明机制的目标值是不停地调整的, 不像比特币那样大约两周才调整一次。

和比特币中最大工作量的链被选为共识区块链不同, 点点币中最大的消耗币龄的链条被选为共识区块链。

另外, 用权益证明机制也使得改变区块链历史的攻击成本降低, 同时“双花”攻击的



成本也会降低。攻击者可以积累一定数量的币龄来强制重组区块链。为了应对这种威胁，点点币引入了一个中心化的 Checkpoint 广播机制，在每天几次广播 Checkpoint 消息，以确认区块链中的交易。确认后的区块不能再被推翻，确认后的区块中的交易将被视为最终交易。

权益证明必须采用某种方法定义任意区块链中的下一合法区块，依据账户结余来选择将导致中心化，例如单个首富成员可能会拥有长久的优势。为此，人们还设计了其他不同的方法来选择下一合法区块。

未币 (NXT) 和黑币 (Blackcoin) 采用随机方法预测下一合法区块，使用公式查找与权益大小结合的最小哈希值，由于权益公开，每个节点都可以合理准确地预计哪个账户有权建立区块详细的。未来币 NXT PoS 机制见 5.5.2 节。

瑞迪币 (Reddcoin) 引入权益速度证明，即鼓励钱币的流动而非囤积。通过给币龄引入指数衰减函数，使得 1 个币的币龄不会超过 2 币月。

### 5.5.2 NXT PoS 机制

与点点币不同，未来币 NXT 是一个完全使用 PoS 权益证明的虚拟货币，同时它也不采用“币龄”的概念<sup>[12]</sup>。NXT 一次性产生 10 亿个币，平均每分钟出一个块。每 10 个块后，交易就采用 SHA-256 和 Curve25519 签名，Curve25519 是椭圆曲线中提供 128 位安全的一种签名机制。它采用椭圆曲线 Diffie-Hellman (ECDH) 密钥协议机制，也是最快的 ECC 曲线之一，同时该曲线是公开的，没有专利保护。

在 NXT 中，三个数值决定哪个账户有资格竞争生成区块，哪个账户获得产生区块的资格，以及在有冲突时，哪个区块链被选为合法区块。这三个数值是：基础目标值、目标值和累积难度值。

1) 基础目标值：为了获得区块产生权，所有活跃的 NXT 账户竞争生成一个小于给定基础目标值的哈希值。这个基础目标值在每个区块都不一样，它是由上一个区块链的基础目标值乘以需要生成一个区块的时间来获得。

2) 目标值：每个账户基于它当前有效的权益来计算它自身的目标值。

$$T = T_b \times S \times B_e$$

其中： $T$  是新的目标值， $T_b$  是基础目标值， $S$  是从上个区块生成以来的以秒为单位的时间， $B_e$  是当前有效的权益。

从公式可以看出，目标值随着上个区块时间戳后的时间流逝而增加，最大的目标值是  $1.537\ 228\ 67 \times 10^{17}$ 。最小的目标值是上个区块基础目标值的一半。

所有竞争生成区块账户的基础目标值和时间都一样，唯一一个与账户相关的参数是余额。

3) 累积难度是通过以下公式从基础目标值算出的。

$$D_{cb} = D_{pb} + 2^{64} / T_b$$

其中： $D_{cb}$  是当前区块的难度， $D_{pb}$  是上个区块的难度， $T_b$  是当前区块的基础目标值。

**区块产生算法：**链上的每个区块有一个区块产生签名参数。为了参与区块产生流程，一个活跃的账户用它的公钥对上个区块的产生签名进行签名，这个生产一个 64 字节的签名，然后对这个签名进行 SHA-256 哈希，结果的头 8 个字节是一个数值，叫作账户的击中值。这个击中值将与当前目标值来比较。如果击中值比目标值小，那么下一个区块就可以产生。从以上公式可以看出，目标值随着时间的推移而增加，那么即使只有几个活跃账户，其中一个必然会生产一个区块，因为目标值会越来越大。这样一来，通过比较一个账户的击中值和目标值，你就能预测任何一个账户生成区块所需的大致时间。由于任意一个节点都可以查询任何一个活跃账户的余额，因此就可以在比较高的精确度下预测哪一个账户会产生下一个区块。

当一个活跃账户赢得生成区块的权利，它会把 255 个尚未确认的交易打包进一个区块，并将这个区块广播到网络作为候选区块。当网络有多个候选区块时，具有最高累积难度值的区块将被选为当前区块链的新区块。

### 5.5.3 Tendermint PoS 机制

Tendermint 区块链针对比特币挖矿机制的能耗问题以及交易确认慢的问题，提出一个不需要挖矿，且能提升交易确认性能，而又在理论上能保证拜占庭容错的 PoS 机制<sup>[13]</sup>。传统的 PoS 机制存在着“无成本作恶问题”(nothing at stake problem)，也就是说，参与共识的节点可以不用担心损失而同时在两个分叉的链上投票。Tendermint 的 PoS 机制采用押金投票机制避免了这个问题，同时也能够在有不超过 1/3 的拜占庭节点存在的情况下保证共识的一致性和正确性。它是第一个能在理论上证明拜占庭容错的 PoS 共识协议。

Tendermint 区块链由 Validator (验证者) 投票来验证交易。验证者需要在投票期中把一定的币作为抵押物，然后通过广播带有签名的投票交易来投票选出下一个区块来参与共识。验证者的投票权重相当于他们抵押的资金。验证者如果要退出共识流程，可以发一个取消抵押的交易来申请取回押金。在验证者发出取消抵押的交易后，抵押的资金还会被锁定一段设定的期限(抵押锁定期)。等该期限过后，验证者才能花取回的押金。一个共识区块需要有超过 2/3 的投票权重的验证者签名投票同意，才能被确认为共识区块。

当一个验证者故意同时投票给两个不同区块，任何其他节点可以发一个短的证据交易，并把该验证者的两个签名投票的冲突交易作为证据附上。这个交易会被记录在区块链上，并将使得作弊的验证者的抵押资金被没收。只要证据在抵押锁定期过期前提交，作弊的验证者就会被惩罚。需要注意的是，这并不能完全避免超过 2/3 的验证者投票产生一个分叉的区块链，因为有可能作弊者已经取回押金并将资金转给其他节点。这种攻击叫长范围双花攻击。用户可以通过在押金锁定期间同步区块链来避免这种攻击。短范围双花攻击发生在攻击者的押金还在锁定期。通过设定验证者获得的酬劳可以提供经济激励，让验证者意识到作弊损失比诚实投票更大。



Tendermint 共识假设网络是部分同步的，它假设网络的延迟有一个上限，同时非拜占庭节点可以访问在一个区块共识期间保持足够准确的内部时钟，这些内部时钟不一定要和全球时间同步。

Tendermint 的共识流程分三个步骤：验证者首先发 Propose（提议）、Prevote（预投票）和 Precommit（预确认），加上另外两个特殊步骤 Commit（确认）和 NewHeight（新高度），如图 5-13 所示。

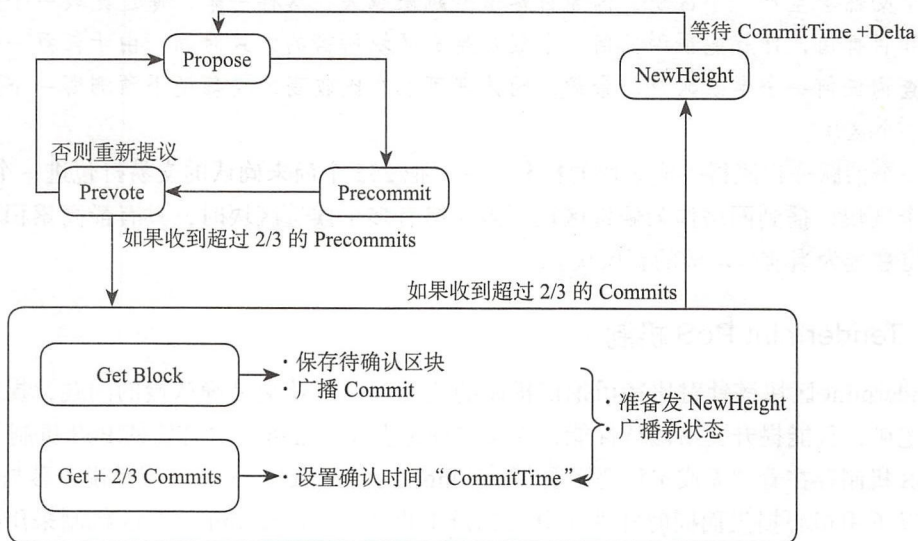


图 5-13 Tendermint 共识流程

在每一轮共识中，验证者会采用轮询机制，按每个验证者押金的比例来选出一个验证者来做提议者，也就是押金多的验证者被抽中做提议者的概率也相应比较大。

共识的第一阶段是提议（Propose）阶段。在提议的开始阶段，选出来的提议者向其他邻节点广播一个提议，收到提议的节点也向它的邻节点广播该提议。

在预投票（Prevote）阶段，每个验证者做决定，如果他在过去的投票阶段有一个提议，但还没被接受并还处于锁住的区块，他可以对那个锁住的 Block 签名并广播一个 Prevote 交易。他如果接受当下这个提议者发出的提议，他也可以对目前的提议签名并广播一个 Prevote 交易。如果他没接到提议，或者接到的是一个不合规的提议，他需要签一个特殊的空 Prevote 交易并广播出去。

在 Precommit 阶段，每个验证者做一个决定。如果验证者收到对他提议的区块超过 2/3 的 Prevotes 投票，他会对该区块签名并广播 Precommit 交易。他同时锁住该区块，并将过去锁住的区块解锁。每个验证节点每次最多能锁住一个提议区块。如果一个提议者收到对他提议的区块超过 2/3 的空 Prevote 投票，他会简单地解锁。当锁或解锁提议区块，需要把 Prevote 交易的这些证据打包，放在一个 proof-of-lock（锁证明）。如果一个提议者没有收到

超过  $2/3$  的 Prevote 或空 Prevote, 那么他不需要签名或锁提议区块。在 Precommit 阶段, 所有的相邻节点互相通报 Precommit 交易。在 Precommit 阶段末, 每个节点会做一个决定。如果节点收到对一个提议区块超过  $2/3$  的 precommit 消息, 那么就进入 Commit 阶段, 否则的话它会继续进入下一轮的提议阶段。即使一个节点还没收到被提议的区块, 它都会进入 Commit 阶段。

Commit 阶段是一个特殊的阶段。有两个并行的条件需要同时满足才能最后确认区块。首先, 节点必须接收到网络 Commit 的提议区块。如果接收到了, 节点需要签名并广播 Commit 消息。然后, 节点必须等收到超过  $2/3$  的 Commit 消息。当这两个条件满足之后, 该节点设一个 CommitTime, 然后转到 NewHeight 阶段。异步通信和本地的 CommitTime 可以让网络即使在不同步的时钟情况下仍然能达成共识。

在区块链高度  $H$  前的 NewHeight 步骤会继续收集在高度  $H1$  的提议区块的 Commit 消息, 直到 CommitTime 加上一个固定的时间过去之后, 这个允许提议的区块包括超过最小  $2/3$  的 Commit 消息, 这样可以把比较慢的验证者的 Commit 消息记录在区块链上。在共识流程的任何阶段, 如果一个节点收到对一个提议区块超过  $2/3$  的 commit 消息。它会立即进入 Commit 阶段。所以有两种进入 Commit 步骤的途径。一是在  $R$  轮对提议区块的确认 (Commit) 投票, 可以算成是对所有  $R_0$  轮 ( $R < R_0$ ) 的 Prevote 和 Precommits。确认 (Commit) 投票消息会在节点之间广播。

在共识流程中的任何时间, 如果一个节点在  $R$  轮在一个提议区块上锁住, 但接收到一个  $R_0$  轮 ( $R < R_0$ ) 的锁证明 (proof-of-lock), 这个节点将解锁。

### (1) Proof of Safety 安全性证明

Safety (安全性) 包含一致性和正确性。也就是要证明其共识机制不会出现分叉。Tendermint 白皮书提供了一个非正式的证明。

如果有低于  $1/3$  的拜占庭投票权重的验证者和至少一个诚实的验证者投票决定  $B$  区块, 那么所有的诚实验证者都会投票决定  $B$  区块。考虑在最早的  $R$  轮, 至少有一个诚实的验证者在  $R$  轮确认  $B$  区块, 这个验证者在  $R$  轮会接收到超过  $2/3$  的 Precommit 消息。假设少于  $1/3$  的是拜占庭节点, 那么至少有  $1/3$  的诚实验证者在  $R$  轮投了  $B$  区块 Precommit。这些诚实的验证者一定会在  $R$  轮对  $B$  区块加锁。其他的区块不可能被确认, 除非一些诚实的验证者解锁  $B$  区块。因为一个验证者只能一次锁住一个区块。也就是说, 不可能出现同时确认  $B$  区块和另一个区块的情况, 因此不可能分叉。

### (2) Proof of Liveness 活性证明

活性指的是在有限的时间内能达成共识, 而不出现死锁或永远达不到一致的情况。Tendermint 给出的非正式证明如下。

如果只有少于  $1/3$  的拜占庭投票权重, 那么协议就不会死锁。只有两个不同的区块被一些不同的诚实验证者在不同的共识轮被锁住 (这个是在没有一个活动的全球攻击者比较少见的情况), 共识流程才能进入死锁的情况。假设某些诚实验证者在  $R$  轮共识锁住了区块  $B$ ,



一些别的诚实验证者在  $R_0$  轮锁住了区块  $B_0$ ，这里  $R < R_0$ 。在这种情况下，在  $R_0$  轮的一个诚实验证者的一个提议所包含的锁证明总会有一个时间解锁，让那些在  $R$  轮锁住  $B$  区块的验证者能继续共识流程。

#### 5.5.4 Ethereum Casper PoS 机制

以太坊在开始的版本规划时期，就计划在未来的 Serenity 版本将共识机制从 PoW 切换到 Casper 的 PoS 共识机制，但后面在制定 Casper 的方案的工作则一拖再拖，直到现在都并没有正式发布。下面从以太坊的一些论坛信息，以及根据以太坊创始人 Vitalik Buterin 和 Virgil Griffith 在网上白皮书《Casper the Friendly Finality Gadget》<sup>[14]</sup> 的资料总结出 Casper 共识机制的一些要点。

出于安全方面的考虑，Casper 的第一版本不会全部替换 PoW 共识算法，而是 PoW 和 PoS 的混合共识算法。具体说就是以太坊会继续使用 PoW 来验证绝大多数区块，PoS 将会每 100 个区块做 checkpoint，也就是永久确认。这个操作给以太坊区块链带来交易确认的最终性。

在 Casper 第一阶段，会在一个 CASPER\_ADDR 地址上发布一个 Casper 合约。这个合约将让所有人打进以太币，并写上一个可用来签名的“验证代码”（可以想象成类似一个公钥），从而成为一个验证者。当一个用户进入活跃的验证池之后，他就可以发消息来参与 PoS 的共识流程。一个验证者的大小指的是他在合约上存放多少以太币。PoS 共识的目的是为了“最终确认”叫作 checkpoints 的关键区块。每第 100 个区块就是一个 checkpoint。为了使一个区块被最终确认，在活跃验证池至少占总验证池  $2/3$  的一个子集验证者（其实是相当于超过  $2/3$  的存储金所代表的投票）需要针对这个区块发 Commit 消息。当一个区块被最终确认后，就意味着永远不能被推翻。即使 99% 的矿工开始支持一个不包含该区块的链，客户都仍然会认为该区块所包含的交易是最终确认的。

验证者广播用自己私钥签名投票消息，投票消息中包含 4 个信息，其中两个是 checkpoint  $s$  和  $t$ 。 $s$  是源 checkpoint，指的是被确认过的 checkpoint， $t$  是目标确认 checkpoint 区块。另外两个是  $s$  和  $t$  所对应的高度  $h(s)$  和  $h(t)$ 。这里的高度指的是从创世区块开始顺着链下来 100 个区块的倍数。因为 Casper 只是用来验证并最终确认每 100 个区块 (checkpoint)，因此这里的高度是指距离创世区块 100 的倍数。

Casper 合约中的规则叫“苛刻条件” (Slashing Conditions)，这些规则用来防止不兼容的两个区块同时被确认，同时防止验证者同时投票确认两个区块。假设验证者广播两个投票消息： $s_1; t_1; h(s_1); h(t_1)$  和  $s_2; t_2; h(s_2); h(t_2)$ ；苛刻条件主要有两个。

第一个是  $h(t_1) \neq h(t_2)$ ，也就是说，一个验证者不能发布两个不同的投票信息，其中的两个 checkpoint 高度一样。第二个是验证者不能发布一个嵌套于另一个 checkpoint 的 checkpoint，也就是说不能发布两个满足下面条件的验证消息，使得： $h(s_1) < h(s_2) < h(t_2) < h(t_1)$ 。违反苛刻条件的验证者会被没收押金。

这两个苛刻条件是为了避免区块链的分叉。如果区块 A 和 B 都是 C 的子块,那么如果 A 和 B 都得到确认,那就说明至少存在着 1/3 的活跃验证池的验证者,同时发了消息确认 A 块和 B 块。这些发双重验证消息的验证者的消息证据会被发到 Casper 合约,他们存的以太币押金的 96% 会被销毁,而 4% 会被奖励给提供证据的举报者。所以一旦推翻一个被确认的区块会带来非常“昂贵”的代价。

Casper 第一阶段主要是实施两部分工作。第一部分是“分叉选择规则”(fork choice rule)。该功能是把过去 PoW 中选用的“最长链”规则改成选用带最大高度的最终确认区块(checkpoint)的规则。第二部分是实现 Casper 验证逻辑的一个后台进程。

### 5.5.5 LPoS 机制

在传统的 PoS 机制中,小份额的权益人有非常小的机会获得出块记账机会。就像在比特币的 PoW 中算力小的矿工非常难挖到矿一样,小的权益人可能需要很多年才能有足够的运气获得出块记账权。这就意味着很多小权益人没有动力运行节点,网络只由小部分的大玩家维持。这样的话,网络的安全性就比较差,因为一个安全的区块链网络需要很多参与者,所以需要激励小额权益者参与出块记账。租借权益证明(Leased Proof of Stake, LPoS)可以解决这个问题。它让权益人将自己的余额租借给别的节点来出块,租借的资金还是由权益人控制,同时在租借到期后可以花掉或者转走。出块后的收益由租借人和出块人共同分享。Waves 是采用 LPoS 的区块链项目。

### 5.5.6 DPoS 机制

PoW 机制和 PoS 机制虽然都能有效地解决记账行为的一致性共识问题,但是现有的比特币 PoW 机制纯粹依赖算力,导致专业从事挖矿的矿工群体似乎已和比特币社区完全分隔,某些矿池的巨大算力俨然成为另一个中心,这与比特币的去中心化思想相冲突。PoS 机制虽然考虑到了 PoW 的不足,但依据权益结余来选择,会导致首富账户的权力更大,有可能支配记账权。DPoS(Delegated Proof of Stake, 股份授权证明机制)的出现正是为了解决 PoW 机制和 PoS 机制的这类不足。

比特股(Bitshare)是一类采用 DPoS 机制的密码货币<sup>[15]</sup>。它期望通过引入一个技术民主层来减少中心化的负面影响。

DPoS 是目前看到最快、最高效,也最去中心化和最灵活的共识模型。DPoS 利用权益人投票的权力来公平民主地解决共识问题。所有的网络参数,从交易费用、生成块的时间以及交易大小,都可以通过选出来的代理人来调整。确定性地选择出块人可以平均 1 秒的速率确认交易。比特股引入了见证人这个概念,见证人可以生成区块,每一个持有比特股的人都可以投票选举见证人。得到总同意票数中的前  $N$  个( $N$  在 Bitshares 中定义为 101,在 EOS 中定义为 21)候选者可以当选为见证人,当选见证人的个数( $N$ )有以下条件确认:至少一半的投票参与者相信  $N$  已经充分地去中心化。



见证人的候选名单每个维护周期（1天）更新一次。见证人随机排列，每个见证人按序有2秒的权限时间生成区块。若见证人在给定的时间片不能生成区块，区块生成权限将交给下一个时间片对应的见证人。DPoS的这种设计使得区块的生成更为快速，也更加节能。

DPoS充分利用了持股人的投票以公平民主的方式达成共识。他们投票选出的 $N$ 个见证人可以视为 $N$ 个矿池，而这 $N$ 个矿池彼此的权利是完全相等的。持股人可以随时通过投票更换这些见证人（矿池），如果他们提供的算力不稳定、计算机宕机，或者试图利用手中的权力作恶。

比特股还设计了另外一类竞选——代表竞选。选出的代表拥有提出改变网络参数的特权，包括交易费用、区块大小、见证人费用和区块区间。若大多数代表同意所提出的改变，持股人有两周的审查期，此期间可以罢免代表并废止所提出的改变。这一设计确保代表技术上没有直接修改参数的权利以及所有的网络参数的改变最终需得到持股人的同意。

2017年新出的TeZoS和EOS项目也是采用DPoS机制。

PoS共识基本上可以分为两种类型。一种类型是用权益模仿PoW机制，通过伪随机数来模仿挖矿决定产生块的权利，这方面的例子有点点币、黑币。该类型还可以是在此基础上的DPoS，它是一个更高效、更公平民主的PoS机制。另一种类型是把拜占庭容错（BFT）和PoS结合起来，可以有严格的数学证明保证只要满足超过 $2/3$ 的诚实参与节点，共识协议不会确认不一致的区块。Tendermint的共识算法是第一个实现BFT的PoS共识算法。以太坊未来的Casper的PoS机制也具有BFT容错能力。

## 5.6 Ripple 共识算法

### 1. Ripple 的网络结构

Ripple（瑞波）是一种基于互联网的开源支付协议，可以实现部分去中心化的货币兑换、支付与清算功能。在Ripple的网络中，交易由客户端（应用）发起，经过追踪节点（tracking node）或验证节点（validating node）把交易广播到整个网络中。追踪节点的主要功能是分发交易信息以及响应客户端的账本请求。验证节点除包含追踪节点的所有功能外，还能够通过共识协议，在账本中增加新的账本实例数据<sup>[16]</sup>。图5-14所示是Ripple共识过程中的节点交互示意图。

### 2. Ripple 共识算法

Ripple的共识达成发生在验证节点之间，每个验证节点都预先配置了一份可信任节点名单，称为UNL（Unique Node List）。在名单上的节点可对交易达成进行投票。每隔几秒，Ripple网络将进行如下共识过程。

1）每个验证节点会不断收到从网络发送过来的交易，通过与本地账本数据验证后，不合法的交易直接丢弃，合法的交易将汇总成交易候选集（candidate set）。交易候选集里面还

包括之前共识过程无法确认而遗留下来的交易。

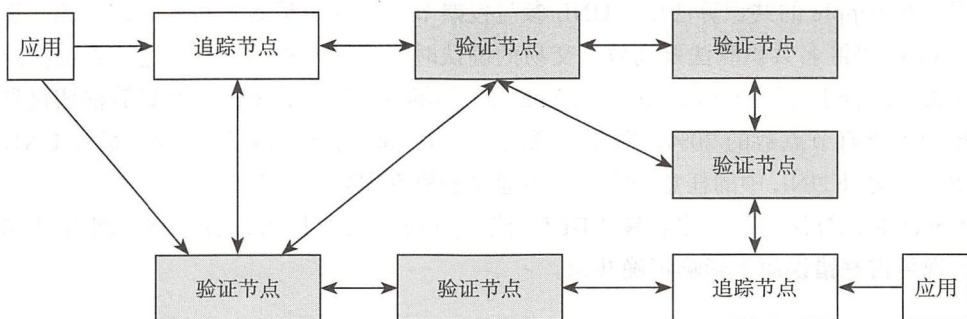


图 5-14 Ripple 共识过程节点交互示意图

2) 每个验证节点把自己的交易候选集作为提案发送给其他验证节点。

3) 验证节点在收到其他节点发来的提案后，如果不是来自 UNL 上的节点，则忽略该提案。如果是来自 UNL 上的节点，就会对比提案中的交易和本地的交易候选集。如果有相同的交易，该交易就获得一票。在一定时间内，当交易获得超过 50% 的票数时，则该交易进入下一轮。没有超过 50% 的交易，将留待下一次共识过程去确认。

4) 验证节点把超过 50% 票数的交易作为提案发给其他节点，同时提高所需票数的阈值到 60%，重复步骤 3、4，直到阈值达到 80%。

5) 验证节点把经过 80% UNL 节点确认的交易正式写入本地的账本数据中，称为最后关闭账本 (Last Closed Ledger)，即账本最后 (最新) 的状态。

图 5-15 所示是 Ripple 共识算法流程。

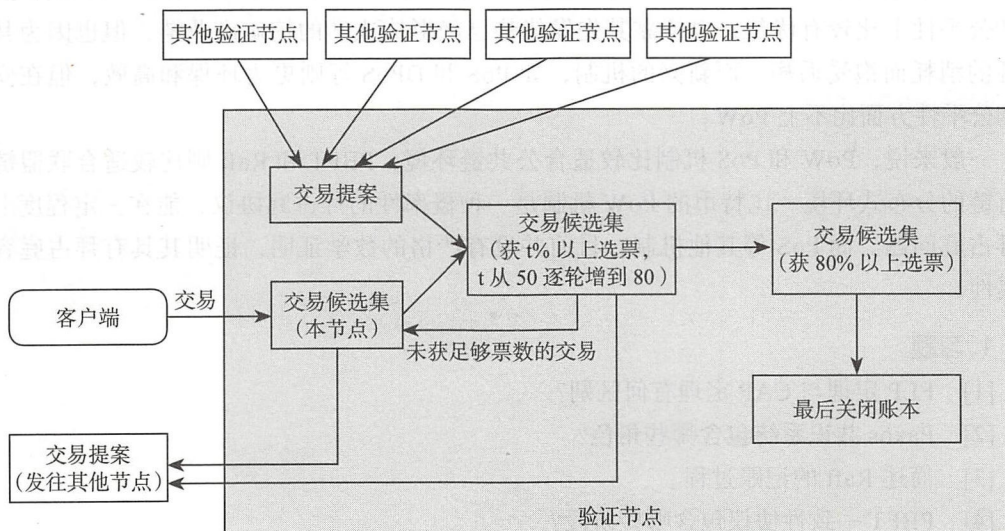


图 5-15 Ripple 共识算法流程



Ripple 共识算法的独特之处在于它不是在全网一次性达成共识，而是在 UNL 子网中达成共识。在 Ripple 的共识算法中，UNL 参与投票节点的身份是事先知道的，因此，算法的效率比 PoW 等匿名共识算法要高效，交易的确认时间只需几秒。当然，这一点也决定了该共识算法只适合于许可链（Permissioned chain）的场景。同时，Ripple 共识算法假设拜占庭节点数少于所有节点数的 20%，需要任意两个 UNL 间重合的节点数至少占最大 UNL 节点数的 20%。另外 UNL 中的任意一个节点串通作恶概率需要小于 20%。

Ripple 共识算法的拜占庭容错（BFT）能力为  $(n-1)/5$ ，即可以容忍整个网络中 20% 的节点出现拜占庭错误而不影响正确共识。

## 5.7 小结

本章主要讨论了共识机制。如何在分布式系统中高效地达成共识是分布式计算领域的重要研究问题，经典的拜占庭容错技术能够在拜占庭服务器不超过 1/3 以及同步通信的情况下，达成拜占庭系统中的共识。而在异步通信情况下，理论上只要有一个拜占庭故障服务器，就无法在全网中达成一致的共识。为了解决实际的分布式一致性问题，很多实用的共识算法被设计出来。这些算法有不同的假设条件，具有不同的优点和局限。本章重点介绍了适用于私有链和联盟链环境的实用拜占庭容错（PBFT）协议，以及针对非拜占庭故障的 Raft 共识算法。Raft 技术则只需故障服务器不超过 50%，就能实现交易记账的共识。

早期的比特币区块链采用高度依赖节点算力的 PoW 机制来保证比特币网络分布式记账的一致性，之后又出现了 PoS 和 DPoS 等共识机制等。除这 3 类主流共识机制外，实际区块链应用中也衍生出了多个变种机制。这些共识机制各有优劣。例如 PoW 共识机制在安全性和公平性上比较有优势，也依靠其先发优势已经形成成熟的挖矿产业链，但也因为其对能源的消耗而饱受诟病。而新兴的机制，如 PoS 和 DPoS 等则更为环保和高效，但在安全性和公平性方面比不上 PoW。

一般来说，PoW 和 PoS 机制比较适合公共链环境，PBFT 和 Raft 则比较适合联盟链和私有链的分布式环境。比特币的 PoW 机制是一种概率性的拜占庭协议，能在一定程度上解决拜占庭问题。而 PoS 等其他机制，目前并没有严格的数学证明，证明其具有拜占庭容错的属性。

### 1. 习题

- [1] FLP 定理与 CAP 定理有何区别？
- [2] Paxos 共识系统包含哪些角色？
- [3] 简述 Raft 的记账过程。
- [4] PBFT 一致性协议包含哪些阶段？
- [5] 简述 PoW 的共识记账过程。

## 2. 参考资料

- [1] Roger Wattenhofer, Strong Consistency, ETH Zurich. Distributed Computing. [www.disco.ethz.ch](http://www.disco.ethz.ch).
- [2] Lamport L, Shostak R, Pease M. The Byzantine generals problem. *ACM Trans. on Programming Languages and Systems*, 1982,4(3):382–401.
- [3] Fischer M.J., Lynch N.A., Paterson M.. Impossibility of distributed consensus with one faulty process. *J. ACM* 32(2): 374–382 (1985).
- [4] Seth Gilbert, Nancy A. Lynch. Perspectives on the CAP Theorem.
- [5] Barbara Liskov and James Cowling. Viewstamped Replication Revisited. MIT Computer Science and Artificial Intelligence Laboratory.
- [6] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169 (1998).
- [7] Ongaro D, Ousterhout J . In Search of an Understandable Consensus Algorithm. In: *Proc. of USENIX Annual Technical Conference 2014*, 305-319.
- [8] Castro M, Liskov B. Practical Byzantine fault tolerance and proactive recovery. *ACM Trans. on Computer Systems*, 2002,20(4):398–461.
- [9] N. Satoshi. Bitcoin: A peer-to-peer electronic cash system. Available on: <http://bitcoin.org/bitcoin.pdf>, 2012.
- [10] Garay J.A., Kiayias A., Leonardos N.. The Bitcoin Backbone Protocol: Analysis and Applications. *IACR Cryptology ePrint Archive 2014*, 765 (2014).
- [11] Sunny King, Scott Nadal. PPCoin, Peer to Peer Cryptocurrency with Proof of Stake, 2012.
- [12] NXT community, NXT Whitepaper, Revision 4 – Nxt v1.2.2 – July 12, 2014.
- [13] Jae Kwon. Tendermint: Consensus without Mining, 2014. [github.com/tendermint/tendermint/wiki](https://github.com/tendermint/tendermint/wiki).
- [14] Vitalik Buterin and Virgil Griffith: Casper the Friendly Finality Gadget.
- [15] Fabian Schuh, Daniel Larimer. Bitshares 2.0: General Overview, Cryptonomex, [Cryptonomex.com](http://Cryptonomex.com)\_Blacksburg (VA), USA.
- [16] David Schwartz, Noah Youngs, Arthur Britto. The Ripple Protocol Consensus Algorithm. Ripple Labs Inc, 2014.



## P2P 网络

作者：郭莹城

前面几章介绍了区块链的基本概念和架构，也深入介绍了区块链核心技术中的密码学和共识算法。本章我们将着重介绍区块链的另一个重要核心技术——P2P 网络。

### 6.1 P2P 网络简介

#### 1. TCP/IP 与 UDP 协议简介

在介绍 P2P 网络之前，我们先来了解一下互联网的通信协议。

首先我们把每个电脑想象成一个人。每个电脑 CPU 型号不同，运行操作系统不同，就相当于人属于不同国家，有不同的语言和不同的文化背景。这些人之间要进行沟通交流时，如果各说各话，鸡同鸭讲，大家就都听不懂，也没办法交流。如果大家都说同一种语言，比如中文普通话或英语，那么就可以明白彼此表达的是什么意思，就能互相交流。电脑也一样，如果电脑要相互交换信息，就要用“同一种语言”，这个“语言”，就是所谓的通信协议。

现在我们来了解一下现代计算机的通信协议：互联网协议族（Internet Protocol Suite，IPS）<sup>[1]</sup>。其实，我们可以把这些协议族当作计算机的一个“语言规范”。电脑有了互联网协议族，就相当于学会了计算机世界的“外语”，大家都用同一种语言，交流将更明确和快捷。

互联网协议族是一个网络通信模型，是一整个网络传输协议组合，为网际网络的基础通信架构。互联网协议族常被通称为 TCP/IP 协议族（TCP/IP Protocol Suite，或 TCP/IP Protocols），简称 TCP/IP。这是因为该协定家族的两个核心协定：TCP（传输控制协议）和 IP（网际协议），为该家族中最早通过的标准。由于网络通信协议普遍采用分层的结构，当

多个层次的协议共同工作时，就类似计算机科学中的堆栈，因此又被称为 TCP/IP 协议栈 (TCP/IP Protocol Stack)。

互联网协议可以分为 OSI 分层模型和 TCP/IP 分层模型。

OSI 模型，即开放式通信系统互联参考模型 (Open System Interconnection Reference Model)，是国际标准化组织 (ISO) 提出的一个试图使各种计算机在世界范围内互连为网络的标准框架，简称 OSI。

所谓的 OSI 分层模型和 TCP/IP 分层模型，你可以想象它就是一门语言的不同解释。比如：美式英语和英式英语，都是同一种语言，虽然有些词汇的表述或意思有些不一样，但大体内容和语法习惯是相同的。

OSI 和 TCP/IP 参考模型如图 6-1 所示。

从图 6-1 我们可以看出，它们描述是同一个协议，只是 OSI 参考模型更加细化，TCP/IP 参考模型是简化版本。现代计算机主流的协议实现都用的是 TCP/IP 参考模型。TCP/IP 参考模型把互联网协议定义为 4 层协议：网络接口层、网络层、运输层、应用层。

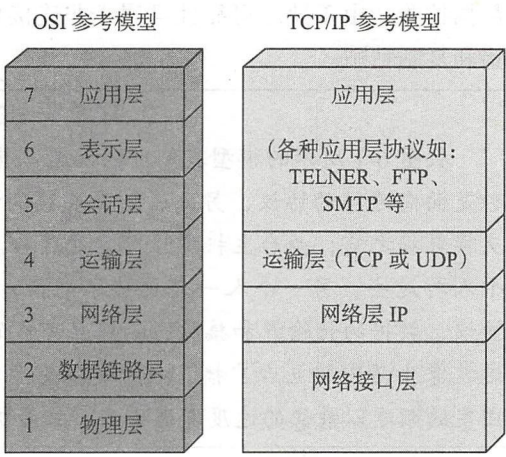


图 6-1 TCP/IP 参考模型

我们从下到上，一一来看不同的协议层。

1) 网络接口层，是 TCP/IP 软件的最底层，负责接收 IP 数据报文并通过网络发送，或者从网络上接收物理帧，抽出 IP 数据报文，交给网络 IP 层。

2) 网络层，负责相邻计算机之间的通信。其功能包括三方面。

① 处理来自传输层的分组发送请求。收到请求后，分组装入 IP 数据报文，填充报头，选择去往目标主机的路径，然后将数据报发往适当的网络接口。

② 处理输入数据报文：首先检查其合法性，然后进行寻径，假如该数据报文已到达目标主机，则去掉报头，将剩下部分交给适当的传输协议；假如该数据报文尚未到达目标主机，则转发该数据报文。

③ 处理路径、流控、拥塞等问题。

3) 传输层，提供应用程序间的通信。其功能包括格式化信息流和提供可靠传输。为实现后者，传输层协议规定接收端必须发回确认，并且假如分组丢失，必须重新发送。

4) 应用层，向用户提供一组常用的应用程序，比如电子邮件、文件传输访问、远程登录等。这层协议主要就是为我们开发或使用的应用程序调用。

综上所述，我们可以想象每一层协议都是特殊的“牛津英文词典”，它们主要的作用就是把应用程序所说的“词汇”翻译成不同的机器语言，最终被计算机理解并执行，从而达到



到信息交流的目的。

P2P 网络大部分都是用 UDP 协议。下面我们简单了解一下 UDP 协议。

从字面上理解，UDP（User Datagram Protocol，用户数据报文协议）是一个简单的面向数据报文（package-oriented）的传输层协议，正式规范为 RFC-768。UDP 只提供数据的不可靠传递，它一旦把应用程序发给网络层的数据发送出去，就不保留数据备份（所以 UDP 有时候也被认为是不可靠的数据报文协议）。UDP 在 IP 数据报文的头部仅仅加入了复用和数据校验。由于缺乏可靠性且属于非连接导向协议，UDP 应用通常允许一定量的丢包、出错和复制粘贴。

---

### TCP 与 UDP 的关系和区别

从图 6-1 所示的模型我们可以知道，其实 TCP 与 UDP 都是传输层的协议，只是它们一种是面向连接的协议，另一种是无连接的协议。TCP 协议是面向连接的协议，也就是说在交换数据之前，两台主机建立可靠的连接。我们可以想象 TCP 协议就像两个人打电话，一个人打过去，另一个人一定要接了电话，两个人才能正常通话。而 UDP 协议是无连接的协议，数据的传输源和接收源不建立可靠的连接，传输源尽可能快速地把信息发送到网络。这就像生活中的电台广播。电台是不需要与每台收音机建立连接的，它只需要把信号通过固定的频率以最快的速度发送出去，收音机只要调到这个频率，就可以接收到声音信号。

---

## 2. P2P 原理简介

我们来想象一个场景。在我们的生活中，每个人都会发短信或微信，但短信或微信都是要经过 WiFi 路由器和电信中心服务器，才能与其他手机用户相互通信。如果有一天，这个 WiFi 信号断了或电信中心服务器停机了，我们就可能发不了短信或微信。那问题来了，有没有可能当 WiFi 信号断了或电信中心服务器停机时，手机用户照样可以互相发信息呢？有，我们可以使用蓝牙等其他通信手段。如果手机与手机不经过电信服务器来直接相互通信，我们就可以认为它们是点对点通信，这就是所谓的 P2P。两个手机通过蓝牙相互通信，就构成最简单的 P2P 网络。

我们想象另一种场景，如果这两个手机不是直接相连，而是通过第三个手机来转发才能进行相互通信，这时候，P2P 网络又是如何运作的呢？或者这两个手机是怎么发现对方的呢？首先，我们假定小明想通过 P2P 网络发消息给小红，但他们距离有点远，超过了蓝牙的最大通信距离，但他们中间刚好有个小芳，小芳的手机刚好可以通过蓝牙跟小明和小红通信，也就是说，小明要发消息给小红，必须通过小芳这个第三方节点。这时候，首先小明的手机会不停地从 P2P 网络发现并查找到合适的第三方节点，一般会以最短通信距离为衡量，直到找到小芳的手机，这个过程就叫作“节点发现”。发现可靠节点后，小明的手机就会跟小芳的手机通信，并通过小芳的手机把他的消息发给小红，这个过程叫作“消息路由”。那么，如果这个时候小芳的手机突然没电了怎么办？此时小明的手机又要重新查找新的第三方节点。如果这个时候小刚出现了，刚好在小明与小红之间可以通过蓝牙通信，小

刚就加入到这个 P2P 网络中，这叫作“节点添加”。而小芳的手机就可以从这个 P2P 网络中直接删除，这叫作“节点删除”。总结一下，P2P 网络通信的原理主要包括：节点添加、消息路由、节点发现、节点删除。

### 3. P2P 网络协议介绍

现在我们来查看 P2P 网络协议。P2P (Peer-to-Peer networking, 对等网络) 可以定义为：网络的参与者共享所拥有的一部分硬件资源（处理能力、存储能力、网络连接能力、打印机等），这些共享资源通过网络提供服务 and 内容，能被其他对等节点 (Peer) 直接访问而无须经过中间实体。在此网络中的参与者既是资源、服务和内容的提供者 (Server)，又是资源、服务和内容的获取者 (Client)。与之相对应的是现在比较流行的中心服务器网络。多个客户端只能通过访问中心服务器才能获得资源、服务和内容。两者的比较如图 6-2 所示。

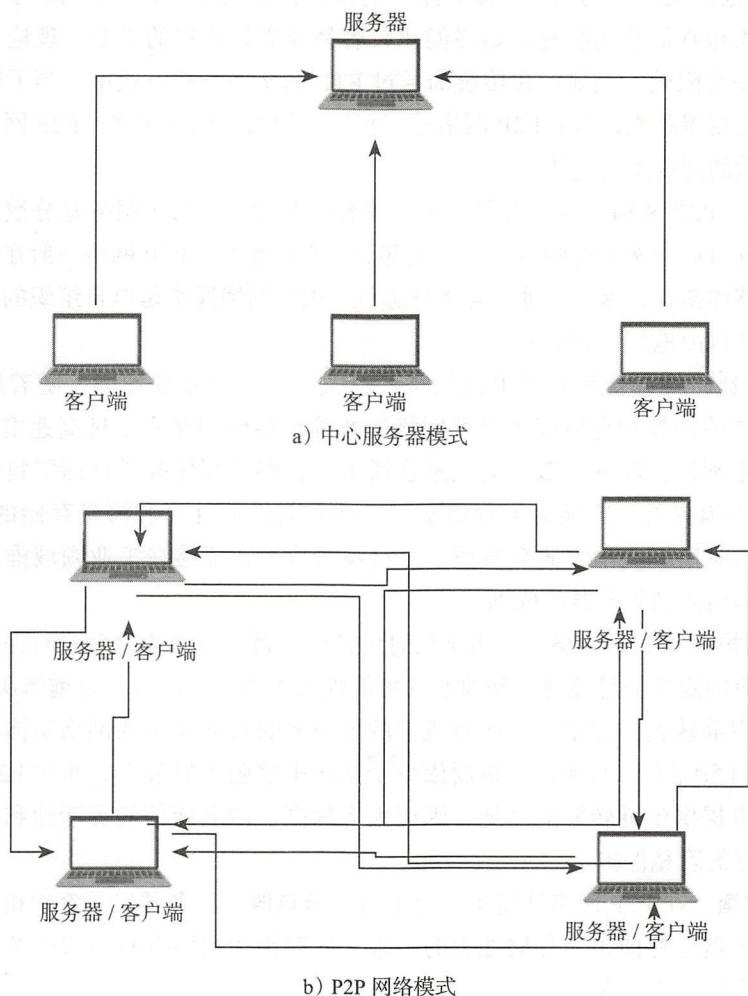


图 6-2 中心服务器模式与 P2P 网络模式比较



P2P 网络也可以称为点对点网络，它不只是一种网络拓扑，其体现了互联网最本质的特性：**去中心化**。互联网起源于美国国防部的研究中心。为了避免敌方摧毁指挥中心，科学家发挥逆向思维，把中心分散到每个电脑上，所谓无中心、去中心。这样，敌方再也没有办法真正摧毁指挥中心了。除非真的将这个星球炸了，否则其中一个基地或电脑受损，并不影响其他基地或电脑继续发挥作用。这种“非中心化”的思想逐渐成为形成互联网最基本的概念。所以说，点对点网络是互联网的本源。

P2P 网络技术的特点体现在以下几个方面。

1) **非中心化**：网络中的资源和服务分散在所有节点上，信息的传输和服务的实现都直接在节点之间进行，可以无须中间环节和服务器的介入，避免了可能出现的瓶颈。P2P 的非中心化基本特点，带来了其在可扩展性、健壮性等方面的优势。

2) **可扩展性**：在 P2P 网络中，随着用户的加入，不仅服务的需求增加了，系统整体的资源和服务能力也在同步地扩充，始终能比较容易地满足用户的需要。理论上其可扩展性几乎可以认为是无限的。例如：在传统的通过 FTP 的文件下载方式中，当下载用户增加之后，下载速度会变得越来越慢；P2P 网络正好相反，加入的用户越多，P2P 网络中提供的资源就越多，下载的速度反而越快。

3) **健壮性**：P2P 架构天生具有耐攻击、高容错的优点。由于服务是分散在各个节点之间进行的，部分节点或网络遭到破坏对其他部分的影响很小。P2P 网络一般在部分节点失效时能自动调整整体拓扑，保持其他节点的连通性。P2P 网络通常是以自组织的方式建立起来的，并允许节点自由地加入和离开。

4) **高性价比**：性能优势是 P2P 网络被广泛关注的一个重要原因。随着硬件技术的发展，个人计算机的计算和存储能力以及网络带宽等性能依照摩尔定理高速增长。采用 P2P 架构可以有效地利用互联网中散布的大量普通节点，将计算任务或存储资料分布到所有节点上。利用其中闲置的计算能力或存储空间，达到高性能计算和海量存储的目的。目前，P2P 在这方面的应用多在学术研究方面，一旦技术成熟，能够在工业领域推广，就可以为许多企业节省购买大型服务器的成本。

5) **隐私保护**：在 P2P 网络中，由于信息的传输分散在各节点之间进行而无须经过某个集中环节，用户的隐私信息被窃听和泄露的可能性大大缩小。此外，目前解决 Internet 隐私问题主要采用中继转发的技术，从而将通信的参与者隐藏在众多的网络实体之中。在一些传统的匿名通信系统中，实现这一机制依赖于某些中继服务器节点。而在 P2P 网络中，所有参与者都可以提供中继转发的功能，因而大大提高了匿名通信的灵活性和可靠性，能够为用户提供更好的隐私保护。

6) **负载均衡**：在 P2P 网络环境下，由于每个节点既是服务器又是客户机，减少了对传统 C/S 结构服务器计算能力、存储能力的要求。同时因为资源分布在多个节点，可以更好地实现整个网络的负载均衡。

#### 4. P2P 网络协议在业界的应用

目前, P2P 计算技术正逐渐应用到军事、商业、政府信息、通信等领域。根据具体应用不同, 可以把 P2P 大致分为以下类型。

- 文件内容共享和下载, 如 Napster、Gnutella、eDonkey、eMule、Maze、BT 等;
- 计算能力和存储共享, 如 SETI@home、Avaki、Popular Power 等;
- 基于 P2P 技术的协同与服务共享平台, 如 JXTA、Magi、Groove 等;
- 即时通信工具, 包括 ICQ、QQ、Yahoo Messenger、MSN Messenger 等;
- P2P 通信与信息共享, 如 Skype、Crowds、Onion Routing 等;
- 基于 P2P 技术的网络电视, 如 PPStream、PPLive、QQLive、SopCast 等。

## 6.2 P2P 网络核心数据结构与算法

### 6.2.1 P2P 网络数据结构与算法

那么, 现在主流的 P2P 算法有哪些呢? 依照节点信息存储与搜索方式的不同, P2P 协议可以分为 2 大类: 非结构化 (Structured) 的系统与结构化 (Unstructured) 的系统。

#### (1) 非结构化 P2P 系统

在非结构化的系统中, 每个节点存储自身的信息或信息的索引 (如指针和 IP 地址)。当用户需要在 P2P 系统中获取信息时, 他们预先并不知道这些信息 (如某个文件) 会在哪个节点上存储。因此, 在非结构化 P2P 系统中, 信息搜索的算法难免带有一定的盲目性, 例如最简单的泛洪式查找 (类似于广播) 和扩展环查找 (从最近的  $n$  个节点开始, 层层转发直到找到目标或超出了跳数的上限为止)。

一些典型的应用采用了一些优化的办法。如在 Gnutella 中采用了等级制的组成结构: 节点被分成超级节点 (Super Node) 和普通节点。普通节点必须依附于超级节点, 每个超级节点作为一个独立的域管理者, 负责处理域内的查询操作。在查找的过程中, 查询首先在域内进行, 失败后才会扩展到超级节点之间。

非结构化系统的优点在于实现结构简单: 无须中央服务器, 节点之间完全平等, 网络的层次是单一的, 而且节点之间无须维护拓扑信息。

#### (2) 结构化 P2P 系统

在结构化 P2P 系统中, 每个节点只存储特定的信息或特定信息的索引。当用户需要在 P2P 系统中获取信息时, 他们必须知道这些信息 (或索引) 可能存在于哪些节点中。由于用户预先知道应该搜索哪些节点, 避免了非结构化 P2P 系统中使用的泛洪式查找, 因此提高了信息搜索的效率。

但是, 结构化 P2P 也引入了新的问题。

首先, 既然信息是分布存储的, 那么如何将信息分布存储在重叠网中的节点上?



其次，由于节点动态地加入和离开重叠网，如何将拓扑的变更信息通知其他节点？

这个时候，就出现了一种叫作 DHT 的技术，它可以很好地解决信息搜索慢、节点信息动态变更等难题。DHT 的全称为分布式哈希表（Distributed Hash Table），是一种分布式存储方法。在不需要服务器的情况下，每个客户端负责一个小范围的路由，并负责存储一小部分数据，从而实现整个 DHT 网络的寻址和存储。我们可以把整个 DHT 网络看作整个城市的地图，每个客户端相当于一张只包括一个街道或街区的小地图，但每个小地图有可能有一定的重合，所有的小地图加在一起就构成了整个城市的全貌。

因此，自从 DHT 协议出现以后，结构化 P2P 的应用得到了快速的发展。目前已经有较多较为成熟的 DHT 协议被提出并且得到了应用。其中比较有代表性的有：缓冲阵列路由协议（CARP）、一致性哈希、Chord、内容寻址网络、Pastry。

## 6.2.2 主流数据结构 DHT 与算法

### 1. DHT 简介

DHT 使用分布式哈希算法来解决结构化的分布式存储问题。分布式哈希算法的核心思想是通过将存储对象的特征（关键字）经过哈希运算，得到键值，对象的分布存储依据键值来进行。具体来讲，大致有以下步骤。

1) 对存储对象的关键字进行哈希运算，得到键值。这样就将所有的对象映射到了一个具体的数值范围中。

2) 重叠网中的每个节点负责数值范围中的特定段落。例如，节点 A 负责存储键值为 8000 ~ 8999 的对象，而节点 B 负责存储的键值为 7000 ~ 7999 的对象。这样就将对象集合分布地存储在所有的节点中。节点可以直接存储对象本身，如文件中的一个片段；也可以存储对象的索引，如该对象所在节点的 IP 地址。

3) 结构化的分布式存储问题解决后，剩下的问题就是用户如何才能找到存储着目标信息的节点。在有着大量节点（如 100 万个）的 P2P 系统中，任何节点都不可能拥有全部的“节点—键值—内容”的对应关系。因此用户获得了键值之后，如何找到该键值对应的节点就被称为“DHT 的路由问题”。

4) DHT 协议必须定义优化的查找（路由）算法来完成这一搜寻工作。不同 DHT 协议之间的区别很大程度上就在于定义了不同的路由算法。

DHT 的应用非常简洁——API 简单到只有一项输入和一项输出。

应用层将数据对象（文件、数据块或索引）通过哈希算法获得键值，将该键值提交给 DHT 后，返回的结果就是键值所在节点的 IP 地址。

图 6-3 显示了这种应用结构。

在这样的支持下，可以开发多种 P2P 的应用程序，如网络存储与文件共享、即时消息、音频 / 视频等。图 6-4 显示了这种应用结构。

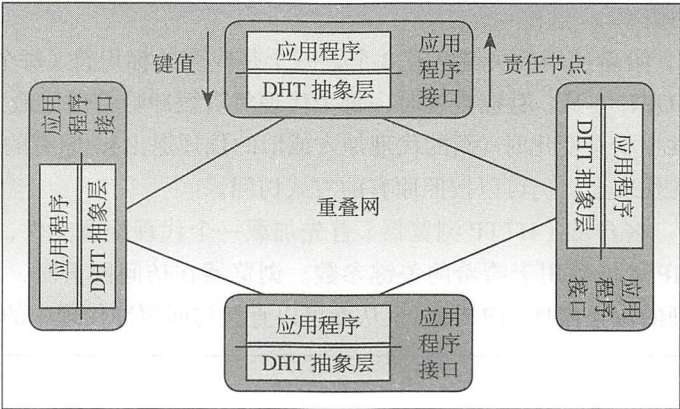


图 6-3 DHT 的应用结构

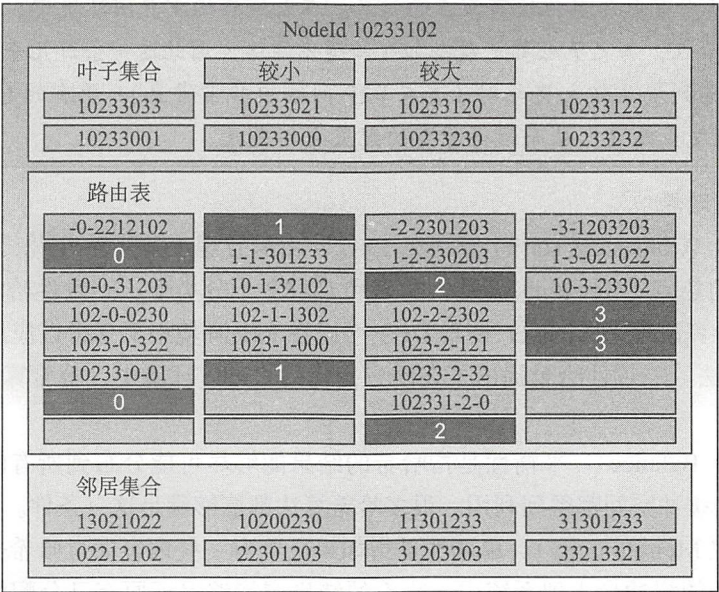


图 6-4 DHT 应用的层次

## 2. 主流 DHT 协议

### (1) 缓冲阵列路由协议

缓冲阵列路由协议 (Cache Array Routing Protocol, CARP) 是由微软公司的 Vinod Valloppillil 和宾夕法尼亚大学的 Keith W. Ross 在 1997 年提出的。该协议可以将 URL 空间映射到一个仅有松散关联关系的 Web Cache 服务器 (在协议中称为“代理”，英文为 Proxy) 阵列中。支持该协议的 HTTP 客户端可以根据要访问的 URL 智能选择目标代理。该协议解决了在代理阵列内分布存储内容的问题，避免了内容的重复存储，提高了客户端访问时



Web Cache 命中的概率。

1) 哈希算法。哈希使用的关键字有 2 个, 一个是代理的标识符 (每个代理均有唯一的标识), 另一个是 URL 本身。存储内容时, 每个代理负责缓冲哈希键值最大的 URL。这样, 当缓冲代理阵列发生少量变化时 (新的代理加入或旧的代理退出), 原有的 URL 还有可能仍然被映射到原来的代理上, 仍可以按照原有的方式访问。

2) 路由算法。客户端 (HTTP 浏览器) 首先加载一个代理配置文件, 该文件中存储了代理的标识符和 IP 地址等用于哈希的关键参数。浏览器在访问网页时, 可以根据 URL 和代理标识获得代理的位置信息 (IP 地址), 从而可以直接访问缓冲代理中的页面。

### 讨论

CARP 的哈希过程比较简单, 路由查找更是简单到最多只有一跳 ( $O(1)$ )。但是 CARP 在 P2P 的应用环境中有一些致命的缺陷。

1) 每个节点必须知道其他所有节点的信息。在大规模的重叠网环境中, 由于可能存在大量的 (数百万) 节点, 加之节点都是动态加入和退出网络, 因此这一条件几乎不可能满足。

2) 在缓冲阵列发生较大变化时 (这在 P2P 网络中非常常见), 原有的 URL 和代理之间的对应关系可能发生改变, 从而使得原有的配置文件失效。

### (2) 一致性哈希

一致性哈希 (Consistent Hash) 算法在 1997 年由麻省理工学院提出<sup>[1]</sup>, 设计目标是为了解决因特网中的热点 (Hot Pot) 问题, 初衷和 CARP 十分类似。一致性哈希修正了 CARP 使用的简单哈希算法带来的问题, 使得 DHT 可以在 P2P 环境中真正得以应用。

1) 哈希算法。一致性哈希提出了在动态变化的 Cache 环境中, 哈希算法应该满足的 4 个适应条件。

① 平衡性 (Balance)。平衡性是指哈希的结果能够尽可能分布到所有的缓冲中去, 这样可以使所有的缓冲空间都得到利用。很多哈希算法都能够满足这一条件。

② 单调性 (Monotonicity)。单调性是指如果已经有一些内容通过哈希分派到了相应的缓冲中, 又有新的缓冲加入到系统中。哈希的结果应能够保证原有已分配的内容可以被映射到新的缓冲中去, 而不会被映射到旧的缓冲集合中的其他缓冲区。

简单的哈希算法往往不能满足单调性的要求, 如最简单的线性哈希。

$$x \rightarrow ax + b \bmod (P)$$

式中,  $P$  表示全部缓冲的大小。不难看出, 当缓冲大小发生变化时 (从  $P_1$  到  $P_2$ ), 原来所有的哈希结果均会发生变化, 从而不满足单调性的要求。

哈希结果的变化意味着当缓冲空间发生变化时, 所有的映射关系需要在系统内全部更新。而在 P2P 系统内, 缓冲的变化等价于 Peer 加入或退出系统, 这一情况在 P2P 系统中会频繁发生, 因此会带来极大的计算和传输负荷。单调性就是要求哈希算法能够避免这一情况的发生。

③ 分散性 (spread) 在分布式环境中, 终端有可能看不到所有的缓冲, 而只能看到其中的一部分。当终端希望通过哈希过程将内容映射到缓冲上时, 由于不同终端所见的缓冲范围有可能不同, 从而导致哈希的结果不一致, 最终的结果是相同的内容被不同的终端映射到不同的缓冲区中。这种情况显然是应该避免的, 因为它导致相同内容被存储到不同的缓冲中去, 降低了系统存储的效率。分散性的定义就是上述情况发生的严重程度。好的哈希算法应能够尽量避免不一致的情况发生, 也就是尽量降低分散性。

④ 负载。负载问题实际上是从另一个角度看待分散性问题。既然不同的终端可能将相同的内容映射到不同的缓冲区中, 那么对于一个特定的缓冲区而言, 也可能被不同的用户映射为不同的内容。与分散性一样, 这种情况也是应当避免的, 因此好的哈希算法应能够尽量降低缓冲的负荷。

从表面上看, 一致性哈希针对的是分布式缓冲的问题, 但是如果将缓冲看作 P2P 系统中的 Peer, 将映射的内容看作各种共享的资源 (数据、文件、媒体流等), 就会发现两者实际上是在描述同一问题。

2) 路由算法。在一致性哈希算法中, 每个节点 (对应 P2P 系统中的 Peer) 都有随机分配的 ID。在将内容映射到节点时, 使用内容的关键字和节点的 ID 进行一致性哈希运算并获得键值。一致性哈希要求键值和节点 ID 处于同一值域。最简单的键值和 ID 可以是一维的, 比如 0000 ~ 9999 的整数集合。

根据键值存储内容时, 内容将被存储到具有与其键值最接近的 ID 的节点上。例如键值为 1001 的内容, 系统中如果有 ID 为 1000、1010、1100 的节点, 该内容将被映射到 1000 节点。

为了构建查询所需的路由, 一致性哈希要求每个节点存储其上行节点和下行节点的位置信息 (IP 地址)。所谓的上行节点, 就是 ID 值大于自身的节点中最小的节点; 而下行节点, 就是 ID 值小于自身的节点中最大的节点。当节点需要查找内容时, 就可以根据内容的键值决定向上行或下行节点发起查询请求。收到查询请求的节点如果发现自己拥有被请求的目标, 可以直接向发起查询请求的节点返回确认; 如果发现不属于自身的范围, 可以转发请求到自己的上行 / 下行节点。

为了维护上述路由信息, 在节点加入 / 退出系统时, 相邻的节点必须及时更新路由信息。这就要求节点不仅存储直接相连的下行节点位置信息, 还要知道一定深度 ( $n$  跳) 的间接下行节点信息, 并且动态地维护节点列表。当节点退出系统时, 它的上行节点将尝试直接连接到最近的下行节点, 连接成功后, 从新的下行节点获得下行节点列表并更新自身的节点列表。同样的, 当新的节点加入到系统中时, 首先根据自身的 ID 找到下行节点并获得下行节点列表, 然后要求上行节点修改其下行节点列表, 这样就恢复了路由关系。

---

## 讨论

一致性哈希基本解决了在 P2P 环境中最为关键的问题——如何在动态的网络拓扑中分布存储和路由。每个节点仅需维护少量相邻节点的信息, 并且在节点加入 / 退出系统时,



仅有相关的少量节点参与到拓扑的维护中。所有这一切使得一致性哈希成为第一个实用的 DHT 算法。

但是一致性哈希的路由算法尚有不足之处。在查询过程中, 查询消息要经过  $O(n)$  步 ( $O(n)$  表示与  $n$  成正比关系,  $n$  代表系统内的节点总数) 才能到达被查询的节点。不难想象, 当系统规模非常大时, 节点数量可能超过百万, 这样的查询效率显然难以满足使用的需要。换个角度来看, 即使用户能够忍受漫长的时延, 查询过程中产生的大量消息也会给网络带来不必要的负荷。

---

下文中讨论的几种 DHT 协议都对路由做出了优化, 提出了各自的算法。

### (3) Chord 协议<sup>[2]</sup>

Chord 在 2001 年由麻省理工学院提出, 其核心思想就是要解决在 P2P 应用中遇到的基本问题: 如何在 P2P 网络中找到存有特定数据的节点。与前两种协议不同, Chord 专门为 P2P 应用而设计, 因此考虑了在 P2P 应用中可能遇到的特殊问题, 这些内容将在路由的部分进行讨论。

1) 哈希算法: Chord 使用一致性哈希作为哈希算法。在一致性哈希协议中并没有定义具体的算法, 在 Chord 协议中将其规定为 SHA-1。

2) 路由算法: Chord 在一致性哈希的基础上提供了优化的路由算法。

在 Chord 中, 每个节点同样需要存储  $m$  个其他节点的信息, 这些信息的集合被称为查询表 (Finger Table)。一致性哈希中的节点同样具有这样的表格。但在 Chord 中, 表格中的节点不再是直接相邻的节点, 它们的间距 (ID 间隔) 将呈  $2^i$  的关系排列 ( $i$  表示表中的数组下标)。这样形成的节点之间路由关系实际上就是折半查找算法需要的排列关系。

在查询的过程中, 查询节点将请求发送到与键值最接近的节点上。收到查询请求的节点如果发现自身存储了被查询的信息, 可以直接回应查询节点 (这与一致性哈希完全相同); 如果被查询的信息不在本地, 就根据查询表将请求转发到与键值最接近的节点上。这样的过程一直持续到找到相应的节点为止。不难看出, 查询过程实际上就是折半查找的过程。

经过 Chord 的优化后, 查询需要的跳数由  $O(n)$  减少到  $O(\log(n))$ 。这样即使在大规模的 P2P 网络中 (例如  $n = 100\,000\,000$ ), 查询的跳数也仅为  $O(8)$ , 每个节点仅需存储 27 个 ( $\log_2 100\,000\,000$ ) 其他节点的信息。

Chord 还考虑到多个节点同时加入系统的情况并对节点加入 / 退出算法做了优化。

---

### 讨论

Chord 算法本身具有如下优点。

① 负载均衡: 这一优点来自于一致性哈希, 也就是一致性哈希中提到的平衡性。所有的节点以同等的概率分担系统负荷, 从而可以避免某些节点负载过大的情况。

② 分布性: Chord 是纯分布式系统, 节点之间完全平等并完成同样的工作。这使得 Chord 具有很高的鲁棒性, 可以抵御 DoS 攻击。



③ 可扩展性: Chord 协议的开销随着系统规模 (节点总数  $n$ ) 的增加而按照  $O(\log n)$  的比例增加, 因此 Chord 可以用于大规模的系统。

④ 可用性: Chord 协议要求节点根据网络的变化动态地更新查询表, 因此能够及时恢复路由关系, 使得查询可以可靠地进行。

⑤ 命名的灵活性: Chord 并未限制查询内容的结构, 因此应用层可以灵活地将内容映射到键值空间而不受协议的限制。

#### (4) 内容寻址网络

内容寻址网络 (Content-Addressable Network, CAN) 在 2001 年由加州大学伯克利分校提出<sup>[3]</sup>。与 Chord 一样, CAN 也是 DHT 的一个变种。

1) 哈希算法: CAN 的哈希算法与一致性哈希有所不同。Chord 中, 哈希得到的键值总是一维的, 而在 CAN 中, 哈希的结果由  $d$  维的笛卡儿空间来表示。 $d$  是一个由系统规模决定的常量。

2) 路由算法: CAN 的路由查询将在  $d$  维笛卡儿空间中进行。

在 CAN 中, 每个节点自身的 ID 经由哈希后得到  $d$  维向量。经过这样的映射后, 整个 P2P 系统将被映射到一个  $d$  维笛卡儿空间中, 每个节点的位置由其自身 ID 决定。CAN 对邻居节点的定义并不要求成  $2i$  的关系排列, 而是改为用在笛卡儿空间上相邻来表示: 在  $d$  维笛卡儿空间中, 2 个节点的  $d$  维坐标中有  $d-1$  维是相等的, 剩余的一维是相邻的节点称为相邻节点。

CAN 中的节点仅存储相邻节点表。由于在  $d$  维的空间中最多有  $2d$  个相邻的节点, 因此节点的相邻节点表最多有  $2d$  个表项。

在查询的过程中, 查询节点首先计算被查询内容的键值 ( $d$  维向量), 然后在节点列表中查找在笛卡儿空间中与该键值最为接近的相邻节点, 找到后向该节点发送查询请求 (这一策略被称为贪婪策略)。查询请求中将携带被查询内容的键值。收到查询请求的节点如果发现自身存储了被查询的信息, 可以直接回应查询节点 (这与一致性哈希完全相同); 如果被查询的信息不在本地, 就根据相邻节点表将请求转发到与键值最接近的节点上。这样的过程一直持续到找到相应的节点为止。在查询过程中, 被查询节点到目标节点的笛卡儿空间距离单调地减少。

如果查询节点或转发节点发现邻居节点表中无法找到可用的下一跳节点, 则采用非结构化 P2P 常用的扩展环搜索 (Expanding Ring Search, 使用无状态、受控的泛洪算法在重叠网中搜索) 以找到合适的 (符合贪婪策略) 下一节点。

经过 CAN 的优化后, 查询需要的跳数由  $O(n)$  减少到均值为  $(d/4)(n_1/d)$  的随机制, 考虑到  $d$  为常数, 这一值可以表示为  $O(n_1/d)$  或  $O(dn_1/d)$ 。

#### 讨论

CAN 和 Chord 的主要区别在于路由算法不同。相比之下, 在节点数量非常大时, CAN





的平均查询跳数要比 Chord 增加得更快。而且 CAN 查询过程中需要的运算量也要高于 Chord。但 CAN 使用的  $d$  为预先设置的常量，因此并不假设系统节点数量。在节点总数动态变化范围很大的系统中，CAN 的相邻节点表结构保持稳定，这在 P2P 的应用中也是很重要的优点。

### (5) Pastry

Pastry 于 2001 年由位于英国剑桥的微软研究院和莱斯 (Rice) 大学提出 (参见<sup>[4]</sup>)。Pastry 也是 DHT 的一个变种。

1) 哈希算法: Pastry 使用一致性哈希作为哈希算法。哈希计算所得的键值为一维 (实际上使用的是 128bit 的整数空间)。Pastry 也没有规定具体应该采用何种哈希算法。

2) 路由算法: 在 Pastry 协议中, 每个节点都拥有一个 128bit 的标识 (Node ID)。为了保证 Node ID 的唯一性, 一般由节点的网络标识 (如 IP 地址) 经过哈希得到。

Pastry 中的每个节点拥有一个路由表  $R$  (Routing table)、一个邻居节点集  $N$  (Neighborhood Set) 和一个叶子节点集  $L$  (Leaf set), 它们一起构成了节点的状态表。

路由表  $R$  共有  $\log Bn$  ( $B = 2^b$ , 为系统参数, 典型值为 16,  $n$  表示系统的节点总数) 行, 每行包括  $B-1$  个表项, 每个表项记录了一个邻居节点的信息 (节点标识、IP 地址、当前状态等)。这样就形成了拥有  $(B-1)\log Bn$  个条目的二维表格。路由表第  $j$  行的表项所记录的邻居节点的 Node ID 前  $j$  个数位和当前节点的前  $j$  个数位相同, 而第  $j+1$  个数位则分别取从 0 到  $B-1$  的值 (除了与当前节点第  $j+1$  数位的值)。这样形成的路由表很类似 IP 路由中最长掩码匹配的算法。参数  $b$  (或  $B$ ) 大小非常关键:  $B$  过大则节点需要维护很大的路由表, 可能超出节点的负载能力, 但路由表大一些可以存储更多的邻居节点, 在转发时更为精确。平均每次路由查找需要的跳数在 Pastry 中计算的结果是  $\log Bn$ , 因此  $B$  的选择反映了路由表大小和路由效率之间的折中。

叶子节点集合  $L$  中存放的是在键值空间中与当前节点距离最近的  $|L|$  个节点的信息, 其中一半节点标识大于当前节点, 另一半节点标识小于当前节点。 $|L|$  的典型值为  $2b$  或者  $2 \times 2b$ 。

邻居节点集合  $M$  中存放的是在真实网络中与当前节点“距离”最近的  $|M|$  个节点的信息。“距离”的定义在 Pastry 中非常类似 IP 路由协议中对距离的定义, 也就是考虑到转发跳数、传输路径带宽、QoS 等综合因素后所得的转发开销 (可以参见 OSPF 等路由协议)。Pastry 并未提供距离信息的获取方法, 而是假设应用层可以通过某种手段 (人工配制或自动协商) 得到信息并配置邻居节点集合。 $|M|$  的典型值为  $2b$  或者  $2 \times 2b$ 。图 6-5 给出了一个 Pastry 节点状态表的例子。

在图 6-5 所示的节点状态表中, 节点本身的 ID 为 10233102。叶子集合中有 8 项, 每一项都代表一个当前节点已知的其他节点的信息。路由表共有  $4 \times 8$  项, 可以看出由上至下节点 ID 重合的位数 (前缀) 不断增加。邻居集合中的节点 ID 由于来源于应用层, 一般没有规



律可循。

Pastry 的路由过程如下。

1) 路由查询消息中携带被查询对象 ID, 又称消息键值。当收到路由消息时, 节点首先检查消息键值是否落在叶子节点集合的范围内。

2) 如果是, 则直接把消息转发给叶子节点集合中节点标识和消息键值最接近的节点; 否则就从路由表中根据最长前缀优先的原则选择一个节点作为路由目标, 转发路由消息。

3) 如果不存在这样的节点, 当前节点将会从其维护的所有邻居节点集合 (包括路由表叶子集合及邻居集合中的节点) 中选择一个距离消息键值最近的节点作为转发目标。

从上述过程可以看出, 每一步路由和上一步相比都更靠近目标节点, 因此这个过程是收敛的。如果路由表不为空, 每步路由至少能够增加一个前缀匹配数位, 因此在路由表始终有效时, 路由的步数至多为  $\log Bn$ 。

| NodeId 10233102  |            |            |            |
|------------------|------------|------------|------------|
| Leaf set         |            | SMALLER    | LARGER     |
| 10233033         | 10233021   | 10233120   | 10233122   |
| 10233001         | 10233000   | 10233230   | 10233232   |
| Routing table    |            |            |            |
| -0-2212102       | 1          | -2-2301203 | -3-1203203 |
| 0                | 1-1-301233 | 1-2-230203 | 1-3-021022 |
| 10-0-31203       | 10-1-32102 | ?          | 10-3-23302 |
| 102-0-0230       | 102-1-1302 | 102-2-2302 | 3          |
| 1023-0-322       | 1023-1-000 | 1023-2-121 | 3          |
| 10233-0-01       | ?          | 10233-2-32 |            |
| 0                |            | 102331-2-0 |            |
|                  |            | 2          |            |
| Neighborhood set |            |            |            |
| 13021022         | 10200230   | 11301233   | 31301233   |
| 02212102         | 22301203   | 31203203   | 33213321   |

图 6-5 Pastry 节点状态表示例

## 讨论

Pastry 的路由利用了成熟的最大掩码匹配算法, 因此实现时可以利用很多现成的软件算法和硬件框架, 可以获得很好的效率。

与 Chord 和 CAN 相比, Pastry 引入了叶子节点和邻居节点集合的概念。在应用层能够及时准确地获得这两个集合的节点信息时, 可以大大加快路由查找的速度, 同时降低因路由引起的网络传输开销; 不过在动态变化的 P2P 网络中如何理想地做到这一点的确有很大的难度。

Pastry 的典型应用包括 PAST<sup>[5][6]</sup> 和 SCRIBE<sup>[7]</sup>。

目前 DHT 算法的发展方向非常多, 不断有新的改进算法被提出来。就笔者目前了解到的信息而言, 至少有以下一些方向。

### (1) 接近性

文中提到的 DHT 算法中, 除了 Pastry 以外, 均未考虑重叠网络拓扑结构与真实的 IP 网络之间的重合关系。节点之间进行对等通信时, 不会考虑优先选取距离自己最近的节点。这样就使得最终形成的重叠网结构混乱、效率低下。因此如何让节点获得并利用接近性 (Proximity) 信息就非常重要。





## (2) 结构化

目前基于 DHT 的应用尚未大规模展开,很多工程上的细节问题尚待解决。例如:目前有很多种类的 P2P 应用,如文件存储和共享、电子邮件、流媒体等。这些应用在处理 P2P 路由算法、拓扑维护和信息检索上使用的方法均有很大差别,导致即使是同类的应用也无法实现互通。如何为各种 P2P 的应用抽象出一个通用的层次,也是目前研究的热点问题之一。

## (3) 信息查询

基于分布式哈希表的查询是一种单关键字的精确匹配,尽管相比非结构化系统它使得系统资源可被确定性地查询到,但它也极大地限制了查询的应用范围。目前有许多改进的结构化查询算法已经被提出来。

### 6.2.3 区块链 P2P 网络协议

现在我们来研究一下主流的区块链技术都是用哪些网络协议。因为经过之前的学习,我们已经了解到区块链是一种去中心化的分布式账本,每个客户端都会维护一个完整的记账记录。这就需要用一种高效、安全、健壮的网络协议来达到同步数据的目的。

现在就从最流行的几个区块链来一一分析,它们到底用的是哪些 P2P 网络协议。

#### 1. 比特币区块链

##### (1) 比特币 P2P 网络与网络节点<sup>①</sup>

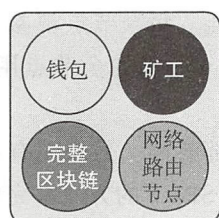
比特币所采用的 P2P 网络架构不仅仅是选择拓扑结构这么简单。比特币被设计为一种点对点的数字现金系统,它的网络架构既是这种核心特性的反映,也是该特性的基石。去中心化控制是设计时的核心原则,它只能通过维持一种扁平化、去中心化的 P2P 共识网络来实现。

“比特币网络”是按照比特币 P2P 协议运行的一系列节点的集合。我们使用“扩展比特币网络”指代所有包含比特币 P2P 协议、矿池挖矿协议、Stratum 协议(stratum 协议是目前最常用的矿机和矿池之间的 TCP 通信协议,详情参见 <https://en.bitcoin.it/wiki/Stratum>),以及其他连接比特币系统组件相关协议的整体网络结构。除了比特币 P2P 协议外,比特币网络中也包含其他协议。例如 Stratum 协议就被应用于挖矿以及轻量级或移动端比特币钱包之中。网关(gateway)路由服务器提供这些协议,使用比特币 P2P 协议接入比特币网络,并把网络拓展到运行其他协议的各个节点。

尽管比特币 P2P 网络中的各个节点相互对等,但是根据所提供的功能不同,各节点可能具有不同的分工。每个比特币节点都是路由、区块链数据库、挖矿、钱包服务的功能集合而成的。一个全节点(完整区块链节点)包括如图 6-6 所示的 4 个功能。

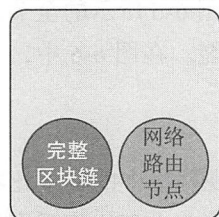
<sup>①</sup> 参见 <http://zhibimo.com/read/wang-miao/mastering-bitcoin/>。





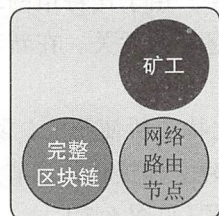
#### 核心客户端 (Bitcoin Core)

在比特币 P2P 网络中, 包含钱包、矿工、完整区块链数据库、网络路由节点。



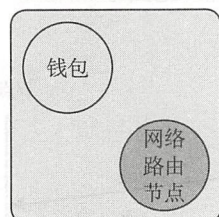
#### 完整区块链节点

在比特币 P2P 网络中, 包含完整区块链以及网络路由节点。



#### 独立矿工

包含具有完整区块链副本的挖矿功能, 以及比特币 P2P 网络路由节点。



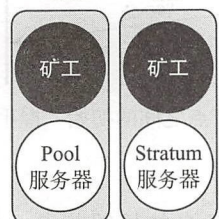
#### 轻量 (SPV) 钱包

包含不具有区块链的钱包以及比特币 P2P 网络节点。



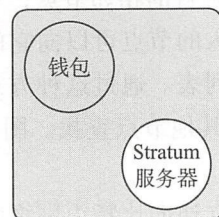
#### 矿池协议服务器

将运行其他协议的节点 (例如矿池挖矿节点、Stratum 节点), 连接至 P2P 网络的网关路由器。



#### 挖矿节点

包含不具有区块链、但具备 Stratum 协议节点 (S) 或其他矿池挖矿协议节点 (P) 的挖矿功能。



#### 轻量 (SPV) Stratum 钱包

包含不具有区块链的钱包、运行 Stratum 协议的网络节点。

图 6-6 完整功能的比特币网络节点





每个节点都参与全网络的路由功能，同时也可能包含其他功能。每个节点都参与验证并传播交易及区块信息，发现并维持与对等节点的连接。在图 6-6 所示的全节点用例中，名为“网络路由节点”的圆圈即表示该路由功能。

全节点能够独立自主地校验所有交易，而不需要借由任何外部参照。另外还有一些节点只保留了区块链的一部分。它们通过一种名为“简易支付验证”(SPV)的方式来完成交易验证。这样的节点被称为“SPV 节点”，又叫“轻量级节点”。在图 6-6 所示的全节点用例中，名为“完整区块链”的圆圈即表示了全节点区块链数据库功能。在图 6-6 中，SPV 节点没有此圆圈，以示它们没有区块链的完整拷贝。

## (2) 网络发现

当新的网络节点启动后，为了能够参与协同运作，它必须发现网络中的其他比特币节点。新的网络节点必须发现至少一个网络中存在的节点并建立连接。由于比特币网络的拓扑结构并不基于节点间的地理位置，因此各个节点之间的地理信息完全无关。在新节点连接时，可以随机选择网络中存在的比特币节点与之相连。

节点通常采用 TCP 协议、使用 8333 端口（该端口号通常是比特币所使用的，除 8333 端口外也可以指定使用其他端口）与已知的对等节点建立连接。在建立连接时，该节点会通过发送一条包含基本认证内容的 version 消息开始“握手”通信过程（见图 6-7）。

网络中的对等节点通过对 verack 消息的响应进行确认并建立连接；有时候，如果接收节点需要互换连接并连回起始节点，也会传回该对等节点的 version 消息。

新节点是如何发现网络中的对等节点的呢？虽然比特币网络中没有特殊节点，但是客户端会维持一个列表，列出那些长期稳定运行的节点。这样的节点被称为“种子节点”(seed nodes)。新节点并不一定需要与种子节点建立连接，但连接到种子节点的好处是可以通过种子节点来快速发现网络中的其他节点。当建立一个或多个连接后，新节点将一条包含自身 IP 地址的 addr 消息发送给其相邻节点。相邻节点再将此条 addr 消息依次转发给它们各自的相邻节点，从而保证新节点信息被多个节点所接收，保证连接更稳定。另外，新接入的节点可以向它的相邻节点发送 getaddr 消息，要求它们返回其已知对等节点的 IP 地址列表。通过这种方式，节点可以找到需连接到的对等节点，并向网络发布它的消息，以便其他节点查找。图 6-8 描述了这种地址发现协议。

节点必须连接到若干不同的对等节点才能在比特币网络中建立通向比特币网络的、种类各异的路径(path)。由于节点可以随时加入和离开，通信路径是不可靠的，因此，节点

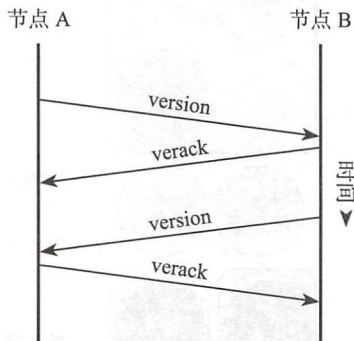


图 6-7 对等节点之间的初始“握手”通信



必须持续进行两项工作：在失去已有连接时发现新节点，并在其他节点启动时为其提供帮助。节点启动时只需要一个连接，因为第一个节点可以将它引荐给它的对等节点，而这些节点又会进一步提供引荐。一个节点如果连接到大量的其他对等节点，这既没必要，也是对网络资源的浪费。在启动完成后，节点会记住它最近成功连接的对等节点；因此，当重新启动后它可以迅速与先前的对等节点网络重新建立连接。如果先前的网络对等节点对连接请求无应答，该节点可以使用种子节点进行重新启动。

如果已建立的连接没有数据通信，所在的节点会定期发送信息以维持连接。如果节点持续某个连接长达 90 分钟都没有任何通信，它会被认为已经从网络中断开，网络将开始查找一个新的对等节点。

因此，比特币网络会随时根据变化的节点及网络问题进行动态调整，不需经过中心化的控制即可进行规模增减的有机调整。

### （3）全节点

全节点是指维持包含全部交易信息的完整区块链的节点。更加准确地说，这样的节点应当被称为“完整区块链节点”。在比特币发展的早期，所有节点都是全节点，当前的比特币核心客户端也是完整区块链节点。但在过去的两年中出现了许多新型客户端，它们不需要维持完整的区块链，而是作为轻量级客户端运行。

完整区块链节点保有完整的、最新的、包含全部交易信息的比特币区块链拷贝。这样的节点可以独立地建立并校验区块链，从第一区块（创世区块）一直建立到网络中最新的区块。完整区块链节点可以独立自主地校验任何交易信息，而不需要借助任何其他节点或其他信息来源。完整区块链节点通过比特币网络获取包含交易信息的新区块更新，在验证无误后将此更新合并至本地的区块链拷贝之中。

运行完整区块链节点可以给用户一种纯粹的比特币体验：无须借助或信任其他系统即可独立地对所有交易信息进行验证。辨别用户是否在运行全节点是十分容易的：只需要查看永久性存储设备（如硬盘）是否有超过 200GB 的空间被用来存储完整区块链即可。如果需要很大的磁盘空间，并且同步比特币网络耗时 2 ~ 3 天，使用的则是全节点。这就是摆脱中心化管理，获得完全的独立自由所要付出的代价。

### （4）交换“库存清单”

一个全节点连接到对等节点之后，第一件要做的事情就是构建完整的区块链。如果该节点是一个全新节点，那么它就不包含任何区块链信息，它只知道一个区块——静态植入在客户端软件中的创世区块。新节点需要下载从 0 号区块（创世区块）开始的数十万区块的

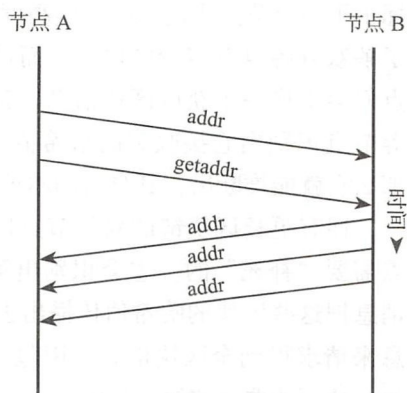


图 6-8 地址广播及发现





全部内容，才能跟网络同步，并重建全区块链。

同步区块链的过程从发送 version 消息开始，这是因为该消息中含有的 BestHeight 字段标示了一个节点当前的区块链高度（区块数量）。节点可以从它的对等节点中得到版本消息，了解双方各自有多少区块，从而可以与其自身区块链所拥有的区块数量进行比较。对等节点们会交换一个获取区块信息，其中包含它们本地区块链的顶端区块哈希值。如果某个对等节点识别出它接收到的哈希值并不属于顶端区块，而是属于一个非顶端区块的旧区块，那么它就能推断出：其自身的本地区块链比其他对等节点的区块链更长。

拥有更长区块链的对等节点比其他节点有更多的区块，可以识别出哪些区块是其他节点需要“补充”的。它会识别出第一批可供分享的 500 个区块，通过使用 inv (inventory) 消息把这些区块的哈希值传播出去。缺少这些区块的节点便可以通过各自发送的 getdata 消息来请求得到全区块信息，用包含在 inv 消息中的哈希值来确认是否为正确的被请求的区块，从而读取这些缺失的区块。

首先它都会从发送 getblocks 消息开始，收到一个 inv 响应；接着开始下载缺失的区块库存清单和区块广播协议，如图 6-9 所示。

#### （5）简易支付验证节点

并非所有的节点都有能力存储完整的区块链。许多比特币客户端被设计成运行在空间和功率受限的设备上，如智能电话、平板电脑、嵌入式系统等。对于这样的设备，通过简化的支付验证（SPV）方式可以使它们在不存储完整区块链的情况下进行工作。这种类型的客户端被称为 SPV 客户端或轻量级客户端。

SPV 节点只需下载区块头，而不用下载包含在每个区块中的交易信息。由此产生的不含交易信息的区块链，大小只有完整区块链的 1/1000。SPV 节点不能构建所有可用于消费的 UTXO 的全貌，这是由于它们并不知道网络上所有交易的完整信息。SPV 节点验证交易时所使用的的方法略有不同，这个方法需依赖对等节点“按需”提供区块链相关部分的局部视图。

打个比方来说，每个全节点就像是一个在陌生城市里的游客，只带着一张包含每条街道、每个地址的详细地图。相比之下，SPV 节点就像是这名陌生城市里只知道一条主干道名字的游客，通过随机询问该城市的人来获取分段道路指示。虽然这两个游客都可以通过实地考察来验证一条街道是否存在，但没有地图的游客不知道每个小巷中有哪些街道，也

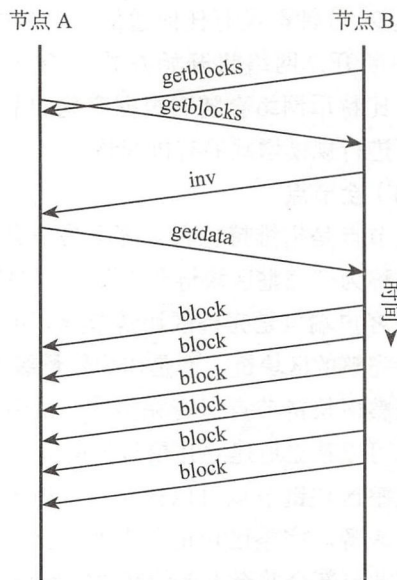


图 6-9 节点通过从对等节点读取区块来同步区块链



不知道附近还有什么其他街道。对他来说,最好的方式就是向更多的人问路,并且希望其中一部分人不是要试图抢劫他。

简易支付验证是通过参考交易在区块链中的深度,而不是高度,来验证它们。一个拥有完整区块链的节点会构造一条验证链,这条链是由沿着区块链按时间倒序一直追溯到创世区块的数千区块及交易组成。而一个 SPV 节点会验证所有区块的链(但不是所有的交易),并且把区块链和相关交易链接起来。

在绝大多数的实际情况中,具有良好连接的 SPV 节点是足够安全的,它在资源需求、实用性和安全性之间维持恰当的平衡。当然,如果要保证万无一失的安全性,最可靠的方法还是运行完整区块链的节点。

SPV 节点使用的是一条 `getheaders` 消息,而不是 `getblocks` 消息来获得区块头。发出响应的对等节点将用一条 `headers` 消息发送多达 2000 个区块头。这一过程 and 全节点获取所有区块的过程没有什么区别。SPV 节点还在与对等节点的连接上设置了过滤器,用以过滤从对等节点发来的未来区块和交易数据流。任何目标交易都是通过一条 `getdata` 的请求来读取的。对等节点生成一条包含交易信息的 `tx` 消息作为响应。区块头的同步过程如图 6-10 所示。

由于 SPV 节点需要读取特定交易从而选择性地验证交易,这样就又产生了隐私风险。与全区块链节点收集每一个区块内的全部交易不同的是,SPV 节点对特定数据的请求可能会在无意中透露了钱包里的地址信息。例如,监控网络的第三方可以跟踪某个 SPV 节点上的钱包所请求的全部交易信息,并且利用这些交易信息把比特币地址和钱包的用户关联起来,从而损害到用户的隐私。

在引入 SPV 节点/轻量级节点后不久,比特币开发人员就添加了一个新功能——Bloom 过滤器,用以解决 SPV 节点的隐私风险问题。Bloom 过滤器通过一个采用概率而不是固定模式的过滤机制,允许 SPV 节点只接收交易信息的子集,同时不会精确泄露哪些是它们感兴趣的地址。

## (6) 交易池

比特币网络中几乎每个节点都会维护一份未确认交易的临时列表,该表被称为“内存池”或“交易池”。节点们利用这个池来追踪记录那些被网络知晓、但还未被区块链包含的交易。随着交易被接收和验证,它们被添加到交易池并通知到相邻节点处,从而传播到网络中。

有些节点的实现还维护了一个单独的孤立交易池。如果一个交易的输入与某未知的交易有关(如与缺失的父交易相关),该孤立交易就会被暂时储存在孤立交易池中直到父交易

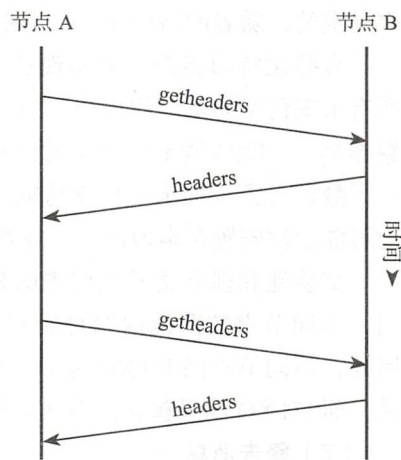


图 6-10 SPV 节点同步区块头



的信息到达。

当一个交易被添加到交易池中，会同时检查孤立交易池，看是否有某个孤立交易引用了此交易的输出（子交易）。任何匹配的孤立交易都会被验证。如果验证有效，它们会从孤立交易池中被删除，并添加到交易池中，使以其父交易开始的链变得完整。对新加入交易池的交易来说，它不再是孤立交易。前述过程重复递归寻找进一步的后代，直至所有的后代都被找到。通过这一过程，一个父交易的到达把整条链中的孤立交易和它们的父级交易重新结合在一起，从而触发整条独立交易链进行级联重构。

交易池和孤立交易池（如有实施）都是存储在本地内存中，并不是存储在永久性存储设备（如硬盘）里。更准确地说，它们是随网络传入的消息动态填充的。节点启动时，两个池都是空的；随着网络中新交易不断被接收，两个池逐渐被填充。

有些比特币客户端的实现还维护一个 UTXO 数据库，也称“UTXO 池”，是区块链中所有未支付交易输出的集合。“UTXO 池”的名字听上去与交易池相似，但它代表了不同的数据集。UTXO 池不同于交易池和孤立交易池的地方在于，它在初始化时不为空，而是包含了数以百万计的未支付交易输出条目，有些条目的历史甚至可以追溯至 2009 年。UTXO 池可能会被安置在本地内存，或者作为一个包含索引的数据库表安置在永久性存储设备中。

交易池和孤立交易池代表的是单个节点的本地视角。取决于节点的启动时间或重启时间，不同节点的两池内容可能有很大差别。相反地，UTXO 池代表的是网络的突显共识，因此，不同节点间 UTXO 池的内容差别不大。此外，交易池和孤立交易池只包含未确认交易，而 UTXO 池只包含已确认交易。

### （7）警告消息

警告消息并不经常使用，但在大多数节点上都有此功能。警告消息是比特币的“紧急广播系统”，比特币核心开发人员可以借此功能给所有比特币节点发送紧急文本消息。这一功能是为了让核心开发团队将比特币网络的严重问题通知所有的比特币用户，例如一个需要用户采取措施的严重 Bug。警告系统迄今为止只被用过几次，最严重的一次是在 2013 年，一个关键的数据库缺陷导致比特币区块链中出现了一个多区块分叉。

## 2. 以太坊区块链

我们知道比特币的目的是建立一个去中心化的 P2P 支付系统，而以太坊的目的跟比特币不同，它是要建立一个去中心化的、永不停机的世界计算机。从这一点来看，以太坊所树立的目标更加宏大，那么它底层通信用的 P2P 网络协议是什么呢？

通过研究以太坊的技术文档（参见 <https://github.com/ethereum/devP2P/blob/master/rlpx.md>）我们发现，它用的就是 DHT 数据结构和类 Kademlia 算法。

那么，什么是 Kademlia 算法呢？

Kademlia 是一种通过分布式哈希表实现的协议算法，它是由 Petar 和 David 为非集中式 P2P 计算机网络而设计的。Kademlia 规定了网络的结构，也规定了通过节点查询进行信息交换的方式。Kademlia 网络节点之间使用 UDP 进行通信。参与通信的所有节点形成一张

虚拟网（或者叫做覆盖网）。这些节点通过一组数字（或称为节点 ID）来进行身份标识。节点 ID 不仅可以用来做身份标识，还可以用来进行值定位（值通常是文件的哈希值或者关键词）。其实，节点 ID 与文件哈希直接对应，它所表示的那个节点存储着从哪里能够获取文件和资源的相关信息。

当我们在网络中搜索某些值（即通常搜索存储文件哈希值或关键词的节点）的时候，Kademlia 算法需要知道与这些值相关的键，然后分步在网络中开始搜索。每一步都会找到一些节点，这些节点的 ID 与键更为接近。如果有节点直接返回搜索的值或者再也无法找到与键更为接近的节点 ID 的时候，搜索便会停止。这种搜索值的方法是非常高效的：与其他的分布式哈希表的实现类似，在一个包含  $n$  个节点系统的值的搜索过程中，Kademlia 仅访问  $O(\log(n))$  个节点。图 6-11 所示为 Kademlia 算法示意图。

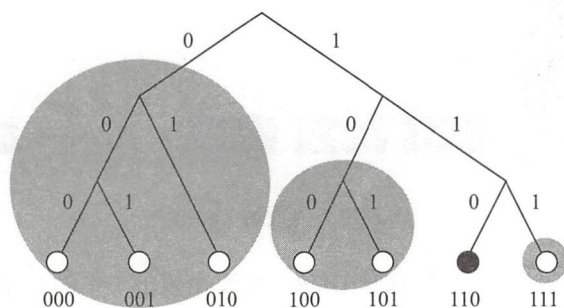


图 6-11 Kademlia 算法示意图

让我们看图 6-11 所示的简单网络：该网络最大可有  $2^3$ ，即 8 个关键字和节点，目前共有 7 个节点加入，每个节点用一个小圈表示（在树的底部）。我们考虑那个用黑圈标注的节点 6，它共有 3 个 K 桶，节点 0、1 和 2（二进制表示为 000、001 和 010）是第一个 K 桶的候选节点，节点 3 目前（二进制表示为 011）还没有加入网络，节点 4 和节点 5（二进制表示分别为 100 和 101）是第二个 K 桶的候选节点，只有节点 7（二进制表示为 111）是第 3 个 K 桶的候选节点。图 6-11 中，3 个 K 桶都用灰色圈表示，假如 K 桶的大小（即 K 值）是 2，那么第一个 K 桶只能包含 3 个节点中的 2 个。

众所周知，那些长时间在线连接的节点未来长时间在线的可能性更大，基于这种静态统计分布的规律，Kademlia 选择把那些长时间在线的节点存入 K 桶。这一方法增长了未来某一时刻有效节点的数量，同时也提供了更为稳定的网络。

当某个 K 桶已满，而又发现了相应于该桶的新节点的时候，那么首先检查 K 桶中最早访问的节点，假如该节点仍然存活，那么新节点就被安排到一个附属列表中（作为一个替代缓存）。只有当 K 桶中的某个节点停止响应的时候，替代 cache 才被使用。换句话说，新发现的节点只有在老的节点消失后才被使用。

根据技术文档 (<https://github.com/ethereum/devp2p/blob/master/rlpx.md>) 和以太坊源码，我们来看看以太坊 Kad 网络是怎么实现的。



以太坊 Kad 网络的路由表是通过称为 K 桶的数据构造而成。K 桶记录了节点 NodeId、distance、endpoint、ip 等信息。以太坊 K 桶按照与 target 节点距离进行排序，共 256 个 K 桶，每个 K 桶包含 16 个节点，如图 6-12 所示。

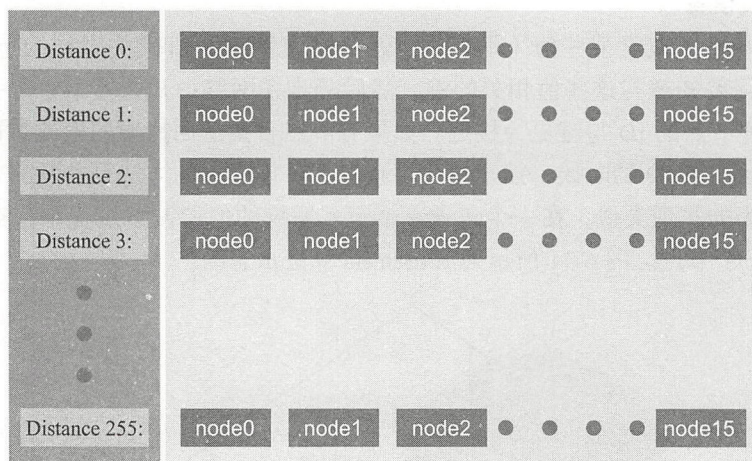


图 6-12 以太坊 Kad 桶

以太坊 Kad 网络中节点间通信基于 UDP，主要由表 6-1 中的几个命令构成，若两个节点间 ping-pong 握手通过，则认为相应节点在线。

表 6-1 Kad 网络通信命令

| 分类       | 功能描述                    | 构成  |
|----------|-------------------------|---|
| PING     | 探测一个节点，判断其是否在线          | <pre>struct PingNode {     h256 version = 0x3;     Endpoint from;     Endpoint to;     uint32_t timestamp; };</pre> |
| PONG     | PING 命令响应               | <pre>struct Pong {     Endpoint to;     h256 echo;     uint32_t timestamp; };</pre>                                 |
| FINDNODE | 向节点查询某个与目标节点 ID 距离接近的节点 | <pre>struct FindNeighbours {     NodeId target; uint32_t timestamp; };</pre>  |

(续)

| 分类        | 功能描述                                     | 构成  |
|-----------|--|---|
| NEIGHBORS | FIND_NODE 命令响应, 发送与目标节点 ID 距离接近的 K 桶中的节点 | <pre> struct Neighbours {     list nodes: struct Neighbour     {         inline Endpoint endpoint;         NodeId node;     };      uint32_t timestamp; }; </pre> |

C++ 版本以太坊源码中, NodeTable 是以太坊 P2P 网络的关键类, 所有与邻居节点相关的数据和方法均由 NodeTable 类实现, 其成员变量和函数如表 6-2 所示。

表 6-2 NodeTable 类

| 成员名称    | 说 明                                      |
|---------|--|
| m_node  | 本节点, 包含 NodeId、endpoint、ip 等             |
| m_state | K 桶, 包含邻居节点的 NodeId、distance、endpoint、ip |
| m_nodes | 已知的节点信息, 但并没有加入 K 桶                      |

表 6-3 是 NodeTable 类的一些详细信息。

表 6-3 NodeTable 类的一些详细信息

| 函数名   | 路 径                                   | 功 能                                  |
|---|---------------------------------------|--------------------------------------|
| NodeTable::NodeTable(bar::io_service& _io, KeyPair const& _alias, NodeIPEndpoint const& _endpoint, bool _enabled) | cpp-ethereum/libp2p/<br>NodeTable.cpp | NodeTable 类构造函数, 初始化 K 桶, 发起邻居节点发现过程 |
| void NodeTable::doDiscovery()   | cpp-ethereum/libp2p/<br>NodeTable.cpp | 具体发现函数                               |
| shared_ptr<NodeEntry> NodeTable::addNode(Node const& _node, NodeRelation _relation)                               | cpp-ethereum/libp2p/<br>NodeTable.cpp | 将节点加入 m_nodes, 并发起 ping 握手           |
| void NodeTable::doDiscover(NodeID _node, unsigned _round, shared_ptr<set<shared_ptr<NodeEntry>>> _tried)          | cpp-ethereum/libp2p/<br>NodeTable.cpp | 底层发现函数, 从 K 桶中选出节点, 发送 FINDNODE 命令   |
| vector<shared_ptr<NodeEntry>> NodeTable::nearestNodeEntries(NodeID _target)                                       | cpp-ethereum/libp2p/<br>NodeTable.cpp | 从 K 桶中选出节点                           |
| void NodeTable::onReceived(UDPSocketFace*, bi::udp::endpoint const& _from, bytesConstRef _packet)                 | cpp-ethereum/libp2p/<br>NodeTable.cpp | Kad 协议处理                             |
| void NodeTable::noteActiveNode(Public const& _pubk, bi::udp::endpoint const& _endpoint)                           | cpp-ethereum/libp2p/<br>NodeTable.cpp | 将新节点加入 K 桶                           |

以太坊 Kad 协议邻居节点发现流程如图 6-13 所示。



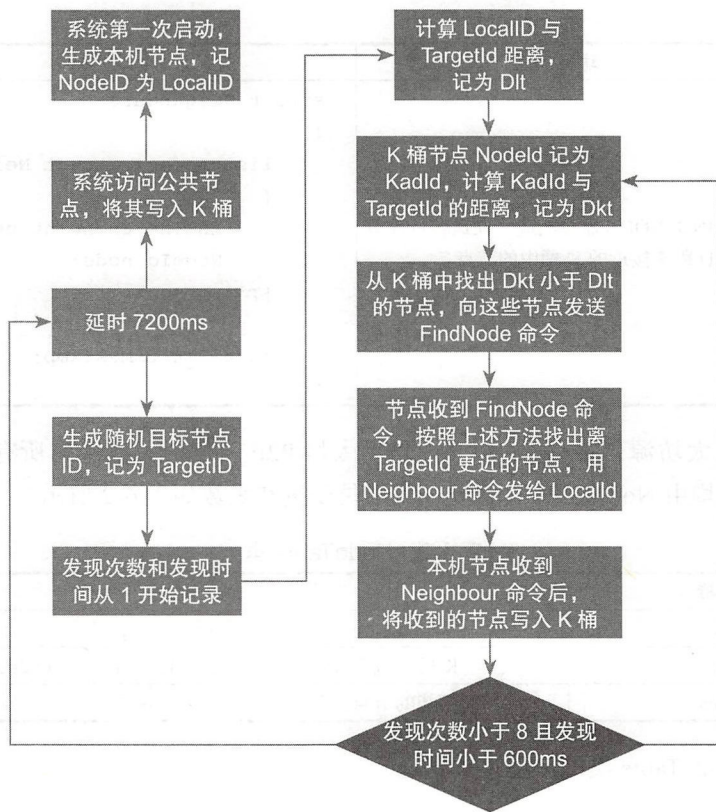


图 6-13 邻居节点发现流程

邻居节点发现流程说明。

1) 系统第一次启动随机生成本机节点 NodeID，记为 LocalID，生成后将固定不变，本地节点记为 local-eth。

2) 系统读取公共节点信息，ping-pong 握手完成后，将其写入 K 桶。

3) 系统每隔 7200ms 刷新一次 K 桶。

4) 刷新 K 桶流程如下：

① 随机生成目标节点 Id，记为 TargetId，从 1 开始记录发现次数和刷新时间；

② 计算 TargetId 与 LocalId 的距离，记为 Dlt；

③ K 桶中节点的 NodeId 记为 KadId，计算 KadId 与 TargetId 的距离，记为 Dkt；

④ 找出 K 桶中 Dlt 大于 Dkt 的节点，记为 K 桶节点，向 K 桶节点发送 FindNODE 命令，FindNODE 命令包含 TargetId；

⑤ K 桶节点收到 FindNODE 命令后，同样执行步骤②~④的过程，将从 K 桶中找到的节点使用 Neighbours 命令发回给本机节点；

⑥ 本机节点收到 Neighbours 后，将收到的节点写入 K 桶中；

⑦ 若搜索次数不超过 8 次，刷新时间不超过 600ms，则返回到②步骤循环执行。

## 6.3 小结

本章介绍了区块链 P2P 网络协议的定义、产生、特征等基本内容，详细阐述了区块链 P2P 技术的基本结构、工作过程、网络构架等内容，最后重点以比特币和以太坊为例，介绍其 P2P 协议的基本内容和工作原理。

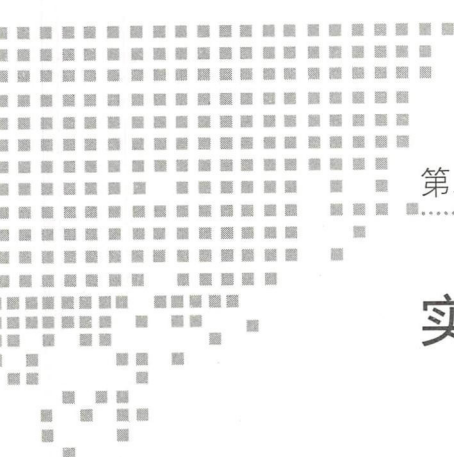
### 习题

- [1] 什么是 P2P 网络协议？
- [2] 什么是 DHT ？
- [3] 主流的 P2P 算法有哪些？
- [4] 比特币区块链 P2P 算法是怎么实现的？
- [5] 以太坊网络中用到的 P2P 算法是哪种算法？

### 参考资料

- [1] David Karger, Eric Lehman, Tom Leighton, Matthew Levine, Daniel Lewin, Rina Panigrahy “Consistent Hashing and Random Trees:Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web”. In Proceedings of the 29th Annual ACM Sym-posium on Theory of Computing (El Paso, TX, May 1997).
- [2] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, Hari Balakrishnan\_ “Chord: A Scalable Peertopeer Lookup Service for Internet Applications” SIGCOMM'01, August 27-31, 2001, San Diego, California, USA.
- [3] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, Scott Shenker “A Scalable Content-Addressable Network” SIGCOMM'01, August 27-31, 2001, San Diego, California, USA.
- [4] Antony Rowstron and Peter Druschel “Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems” Appears in Proc. of the 18th IFIP/ ACM International Conference on Distributed Systems Platforms (Middleware 2001). Heidelberg, Germany, November 2001.
- [5] P. Druschel and A. Rowstron. PAST: A large-scale, persistent peer-to-peer storage utility. In Proc. HotOS VIII, Schloss Elmau, Germany, May 2001.  
<http://www.cs.umd.edu/class/spring2011/cmsc818k/Lectures/Pastry-Past.pdf>.
- [6] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In Proc. ACM SOSP'01, Banff, Canada, Oct. 2001.
- [7] A. Rowstron, A.-M. Kermarrec, P. Druschel, and M. Castro. Scribe: The design of a large-scale event notification infrastructure. Submitted for publication. June 2001.  
<http://www.research.microsoft.com/antr/SCRIBE/>.





## 第二篇 *Part 2*

# 实 战 篇

- 第7章 比特币
- 第8章 以太坊
- 第9章 超级账本 Fabric

# 比特币

作者：蒋 勇

前面几章我们介绍了区块链的基础概念和架构，并对区块链的核心技术，像密码学、共识算法和 P2P 网络做了详细的介绍，下面我们介绍主流区块链平台中的比特币。

比特币是区块链技术的第一个应用，也是到目前为止最落地的应用。事实上，区块链技术的概念就是通过比特币带到世人眼前的——在比特币的技术白皮书中提到了区块数据存储以及一个个区块顺序链接的技术概念。具体内容大家可以参见《比特币：一种点对点的电子现金系统》一文。虽然目前关于比特币的交易和数字货币的法律地位很有争议，但是站在技术层面和学习的角度，是很值得去好好研究一番的。本章就是对比特币各项概念的一个综合阐述，通过学习本章内容，大家可以建立起对比特币的完整理解。

## 7.1 比特币的特点

比特币作为一个区块链技术的应用，自然也具备了区块链系统的基本特点。作为第一个落地应用，且是定义为电子现金系统的应用，其结构设计精巧，运行逻辑简单可靠。具体来说，比特币具备如下的特点。

### （1）分布式网络结构

比特币是一个点对点的网络系统。在由无数比特币节点组成的网络结构中，不存在任何一个地位特殊的服务器节点，也就是说每一个运行节点的地位都是相当的、平等的；节点之间不存在任何依赖关系。在比特币节点网络中，节点之间除了通过“发现协议”来与其他邻近节点进行数据交换和同步外，主要的验证处理，包括交易事务的执行等都是各自独立的。这是一个很大的特点，每一个比特币节点的工作都是保持独立的，并不需要



过多地与其他节点交互。这种结构使比特币网络的单点故障防护能力大大提高。

需要注意的是,上述节点是指比特币全节点,是包含完整的数据账本以及网络路由的,而不是指某些单独的挖矿节点或者钱包节点。我们通常所说的比特币节点指的是全节点,它是一个整体。

## (2) UTXO 账户模型

UTXO, 全称 Unspent Transaction Output, 翻译过来是“未花费交易输出”的意思, 初听起来是比较拗口的。但这是一种很有意思的账户结构设计, 理解它得从了解比特币交易事务的数据结构开始。

在比特币中, 用户可以使用自己的私钥签发一笔转账交易; 然后这笔交易数据就会在网络中进行广播, 直到被矿工节点打包进区块; 然后区块再广播到全网节点, 最终大家都记下这笔账。在整个过程中, 交易数据都是公开传播的。怎么确定这笔公开传播的数据具体是谁签发, 又是转给谁的呢? 在比特币的交易事务结构中, 这是通过巧妙地使用公钥系统的非对称加密算法来实现的。在一笔交易事务中包含着转账人的私钥签名, 这是用以识别发送方的, 同时也包含着接收人的公钥 (通常是使用通过公钥的哈希处理, 也就是钱包地址)。配合输入脚本和输出脚本程序, 就构成了一笔可以在比特币网络中进行自动化验证合法性的交易数据。我们来看一下示意图, 如图 7-1 所示。



图 7-1 比特币交易数据关系图

如图 7-1 所示就是比特币中的交易数据关系图。我们可以看到, 其中的每一笔交易都

是由输入和输出两部分组成，环环相扣。除了 Coinbase 交易，普通的转账交易，其输入部分都是来自于上游的输出部分。也就是说，对于某一个账户地址来说，能有多少比特币可以转账，取决于他拥有多少可用的剩余的“输出”中的余额。以图 7-1 来说，002 号交易中总共转出了 12.5 个比特币，其来源就是前一次的交易输出中的余额。在比特币系统中，一个账户地址拥有的所有可用的剩余“输出”就是 UTXO，这就是“未花费交易输出”这个名词的含义所在。这种结构的设计也增强了比特币其他方面的特性，比如不可篡改性。配合工作量证明的共识算法，它还能防止“双花”问题。

### （3）工作量证明

工作量证明机制是比特币中设计使用的一种共识算法。根据 FLP 理论，在网络可靠、存在节点失效的最小化异步模型系统中，不存在一个可以解决一致性问题的确定性算法。如果比特币是设计为一种传统的服务器—客户端结构的话，问题会比较容易解决，程序可以去服务器进行一致性的判断校验。可是这样的话，一旦节点网络的规模越来越大，服务器会不堪重负，整个系统的风险性很大，而且为了维持特定的服务器，网络结构就不再是点对点这种可以随时横向扩展的了。

一直以来，这个问题的解决都是一个难点，而对于要建立在分布式网络环境之下的数字货币系统，更是难上加难。节点之间的数据不能保持一致，对于一个现金系统来说是致命的。

比特币设计者创造性地发明了一种竞争机制。通过节点所在计算机的算力来计算符合某个难度预期的值，谁先计算出谁就拥有记账权，获得记账权的节点将会打包网络中等待确认的交易数据，打包成区块后再广播给其他节点，并等待全网确认，以此来尽可能地保持节点之间账本的一致性。

这是一个天才的设计，非常有创意。看上去很简单，然而却能最大限度地解决异步网络中数据一致性的问题。而且因为这是一种竞争机制，也就是说过程中如果数据有暂时不一致的情况发生，只要等待一段时间，整个网络就会决胜出一条为大多数节点所共识的主链数据。

### （4）数据不可篡改

一直以来，在所有的软件系统中，数据不可篡改几乎是一个伪命题。我们都知道，即便是有权限设计的系统，对于系统管理员也是形同虚设的。只要有密码，或者即使没有密码而通过某种手段绕到后台，还是一样能修改数据的。很难有一种技术方案可以做到很好的防护。

比特币作为一种点对点电子现金系统，如果数据能被篡改那不是致命的事么？这里就要涉及比特币系统设计的另一个强大的结构了，那就是区块链账本结构以及工作量证明机制。比特币的交易事务是存储在一个个的区块中的，各区块之间通过区块哈希值进行链接，形成一个链的结构。大家注意，一旦区块确定下来，区块的哈希值也就固定下来了，如果再修改区块中的数据，哈希值就会发生变化。也就是说，如果有人要想修改某个区块中的



数据,那就意味着哈希值也变化了,从而后续所有的区块都要发生变更才行。那么,能不能去变更后续所有的区块呢?在目前的比特币网络中,这几乎是不可能做到的,因为即便你做了修改并广播出去,其他节点也是不能验证通过的,也就无法在网络中达成共识。

这样的机制是非常巧妙的:整个机制都是公开透明的,没什么要保密的,也没什么特别的权限设计,充分地利用了链的结构以及分布式网络共识机制就实现了不可篡改性。当然了,大家仍要注意,这种不可篡改性是有前提的,那就是网络必须是足够广泛的。如果比特币节点都控制在一家手里,那就谈不上“不可篡改”了。

### (5) 无法伪造

这里的“无法伪造”是指在比特币网络中,要想得到比特币,要么靠挖矿获得,要么靠其他地址的转账,自己是无法凭空伪造一个比特币的。通过上述内容的阐述,我们了解到比特币中的账户模型是 UTXO,也就是说某个账户地址下拥有的可花费的比特币余额是以一个交易输出的形式存在的,那么如果要给自己伪造一个比特币的话,就得构造能让系统识别的交易事务出来。那么问题就来了:如果是伪造一条挖矿所得的交易事务,也就是 Coinbase 交易,那得有符合挖矿难度值的证明;如果是伪造一条其他地址的转账交易,那得保证其他地址中有足够的余额,并且要使用其他地址所对应的私钥来签名这笔交易。很显然,这两者都很难做到,只要条件不具备,发送到网络中的交易数据就会被其他节点拒绝并丢弃。因此,在比特币网络中要想伪造一个比特币是很困难的。

## 7.2 比特币的 P2P 网络

### 7.2.1 点对点的钱包节点分类

#### 1. 拥有全部链数据的全节点

我们知道比特币在技术本质上是一个网络软件。在这个网络软件的组成上,包含了几个重要的部分,如图 7-2 所示。

这些组成部分可以是在一个整体中,也可以是分开运行的。目前通过比特币核心的官网下载的客户端,除了不包括挖矿功能外,集成了其余所有的功能。在所有组成部分中,我们可以看到一个叫“完全节点”的,这是什么意思呢?这是指在这个节点的本地存储中拥有最完整的账本数据,这样的节点就叫完全节点。

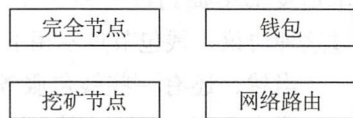


图 7-2 比特币的节点组成

完全节点是比特币网络的骨干组成,其数量的多少以及分布的广泛程度,对比特币网络的安全性、可用性等是有影响的。在完全节点上,对于任何比特币的交易数据以及区块等,都是可以进行最全面完整的验证的,而且在节点本机就可以独立地验证。

有一点大家要注意,建立在完全节点基础之上的比特币钱包也叫全节点钱包,这种钱

包在日常使用并不多，因为要携带着巨大的副本数据。

## 2. 裁剪模式保留部分链数据的轻节点

完全节点中保留了所有的区块链账本数据，时间一长自然会占据相当大的存储空间，便携性不是很好，而且如果不是每天都保持同步的话，从零开始同步或者过了很长一段时间再同步，往往要等很长时间，很不方便。因此，比特币的精简模式就应运而生。简单来说，就是将时间比较长远的、老旧的交易数据以及那些已经没有余额的交易输出都忽略掉或者说裁剪掉，不再保留在本地。

比特币核心客户端从 0.11.0 版本开始支持这种裁剪模式。要启用精简模式，可以在启动比特币客户端时指定 `prune` 参数。比特币的两个客户端 `bitcoin-qt` 和 `bitcoind` 都支持，命令如下：

```
Bitcoin-Qt -prune = 1000
```

或者

```
bitcoind -prune = 1000
```

如果不使用命令参数的话，创建在 `bitcoin.conf` 文件中也是可以的，增加配置项 `prune=1000` 即可。这里的 1000 是可以自己定义的，这个值的作用是设定保存区块和回溯文件的硬盘空间，设置的值为最低值，也就是最少保留这么多。大家可以根据自己的需要去设置。

需要注意的是，设置精简模式后，客户端仍然是会同步所有区块的，只是不会全部保存下来。同时精简模式也是有不少限制的，由于没有保存最完整的账本数据，在完全溯源的时候，会查不到早期的来源数据，但是并不影响正常使用。

## 3. 没有链数据的轻钱包节点

对于钱包来讲，其主要功能无非是保管自己的比特币余额，并且可以转账给其他账户地址，或者接收其他地址的转入。核心客户端带有的钱包是基于完全账本数据的，也就是要正常使用就得带着那一大块的数据。目前已经有一百几十个 GB 了，相当不方便。在比特币技术白皮书中提到过一种 SPV 方式，即 Simplified Payment Verification（简单支付验证）。通过这种协议，钱包节点不用下载链数据，只需要同步区块头即可，这样数据量要小很多。

当然，还有一些钱包服务商提供了更轻量级的客户端。用户完全不用自己来同步任何区块账本数据，只要连接服务商的服务器节点就可以了。这种方式使钱包客户端更加便携。不过在这种方式下，用户一定要注意自己的私钥是否会存储到服务商的服务器上，要注意私钥安全。

## 7.2.2 全节点的分布式存储

### 1. 比特币全节点全球地域分布

比特币迄今已经存在 10 年了。多年的发展也使得其节点的分布相当广泛，可以说是目



前所有区块链系统中节点最为广泛的了。我们来看一下全球节点分布图，如图 7-3 所示。

图 7-3 是 2018 年 3 月从 bitnodes 网站上截取的比例分布图。从图中可以看到，美国、中国和德国的占比是比较大的。

## 2. 全节点的版本及之间的兼容性

比特币是一个开源软件，源码目前托管在 GitHub 之上。多年以来，比特币的源码更新从未停止过，其中不少的功能升级都来自于 BIP (Bitcoin Improvement Proposals, 比特币改进建议)。BIP 是开源社区对比特币发展的功能建议，很多重大的功能升级都能在 BIP 中找到论述，比如对多重签名的支持、确定性钱包的支持、隔离见证的实现等。

随着软件源码的修改维护以及功能升级，比特币的版本也在不断迭代。通过查看 GitHub 上的记录，可以看到比特币已经有 192 个 release 版本，当然这个数字还在增加，截至写作时的最新发布版本是 0.16.0。你没看错，到现在都还没有 1.0 版呢，这都要 10 年了，这也充分地说明了社区的活跃程度，各种新的思想和设计不断涌现，一切都要去尝试。

各个比特币的全节点版本之间并非是完全兼容的，比如多重签名到 2012 年才开始支持，它所产生的多重签名交易，对于仍旧在使用之前版本的节点自然是不能兼容的。因此大家在使用具体的比特币版本时，注意查看支持的功能，不过通常新版本都是支持老版本的协议的。

对于那些后来在比特币的基础上修改基础协议到释出的分叉版本这里不做讨论。事实上，分叉（主要是指硬分叉）后的版本已经不属于比特币核心的全节点版本了。

## 3. 全节点的版本变化历史

到目前为止，比特币已经发布了相当多的版本了，这些在 GitHub 上也可以查看到。每一次大的版本升级都会带来一些新的功能和特性支持。我们来看看几次大的版本变化历史。

- 2009 年 1 月 9 日，0.1.0 版发布，比特币诞生；
- 2011 年 9 月 23 日，0.4.0 版发布，支持钱包私钥的加密；
- 2011 年 11 月 21 日，0.5.0 版发布，使用 Qt 库完全支持了图形界面；
- 2012 年 3 月 30 日，0.6.0 版发布，支持 33 字节的压缩公钥替代之前的 65 字节公钥；
- 2013 年 2 月 19 日，0.8.0 版发布，支持 LevelDB 数据库存储索引及布隆过滤器；
- 2017 年 9 月 14 日，0.15.0 版发布，支持生成隔离见证地址；

| 12221 NODES   |                    |               |
|---|--------------------|---------------|
| 24-hour charts »  |                    |               |
| Top 10 countries with their respective number of reachable nodes are as follow. |                    |               |
| RANK  | COUNTRY            | NODES         |
| 1   | United States      | 2690 (22.01%) |
| 2   | China              | 2310 (18.90%) |
| 3   | Germany            | 1960 (16.04%) |
| 4   | France             | 703 (5.75%)   |
| 5   | Netherlands        | 481 (3.94%)   |
| 6   | United Kingdom     | 402 (3.29%)   |
| 7   | Canada             | 394 (3.22%)   |
| 8   | Russian Federation | 360 (2.95%)   |
| 9   | n/a                | 295 (2.41%)   |
| 10  | Singapore          | 235 (1.92%)   |
| More (100) »  |                    |               |

图 7-3 全球节点分布图

□ 2018 年 2 月 26 日, 0.16.0 版发布, 全面支持隔离见证钱包操作。

这里只是列举了部分历史, 但大家可以看到, 比特币的代码升级和功能完善是一直在进行的。

### 7.2.3 交易和区块在节点间的传播同步

#### 1. 交易的内容生成发布验证及传播

比特币网络中主要处理的就是交易事务。一笔交易从签发开始, 然后广播到网络中, 网络中的其他节点会接收交易数据并且进行验证, 验证通过后会继续广播到网络中的其他节点。通过这种“病毒式”的传播方式, 信息很快就能让大多数的节点都接收到。当然, 在这个阶段, 交易事务还没有被写入到区块中, 就是还没有上链, 等到被挖矿节点打包进区块后会再次同步给其他节点, 直到被全网中绝大部分的节点写入自己本地的账本, 即是得到了全网的共识。

交易的创建可以由某个账户的私钥所有人进行直接签发, 也可以由一个代理人创建后提交给私钥所有人进行签发, 构造完毕后的交易数据必须是有效的。在发送到网络中时, 对网络环境也没有特殊的要求, 对于传输的交易数据是否要加密也没有“必须”的要求。其他节点接收到后进行独立的验证, 验证过程并不需要与签发者发生任何交互。

#### 2. 矿池将交易打包成区块广播

在比特币诞生的早期, 挖矿主要是 solo 形式, 也就是个体矿工独立工作。这主要是因为初期参与挖矿的节点还不多, 挖矿的难度也不大, 甚至在很长一段时间内都没什么难度变化。比特币诞生初期, 中本聪自己一个人就默默地挖了一年。然而随着参与者越来越多, 难度越来越大, 个体矿工挖到矿的希望越来越渺茫, 这个时候就出现了多个矿工联合挖矿的“矿池”模式。

矿池实际上是一个服务程序, 矿工们将自己的矿机通过专用的协议连接到矿池, 并且在矿池服务器注册账号, 矿池就可以来协调成百上千个矿工一起挖矿。矿池会将每一次需要运算的任务分配给注册进来的矿机, 由于数量众多的矿机分担了巨大的运算任务, 因此能够更快地计算出目标值, 也就是挖到矿。矿池程序通常也连接着完全节点, 通过完全节点可以获得需要打包的交易事务。挖矿成功后, 就会打包成区块, 然后进行网络广播。我们可以看到, 矿池程序和普通的个体挖矿节点, 虽然方式不同, 但过程还是类似的。

#### 3. 全节点对区块的验证及传播同步

通过上面的阐述我们已经知道, 比特币中的全节点主要是指保留完整区块链账本数据的完全节点, 但是全节点不一定是具备挖矿功能的。事实上, 目前比特币的挖矿都是独立运行的, 当然了, 挖矿程序是离不开全节点的。全节点是组成比特币网络的骨干节点, 如果在比特币网络中, 全节点的数量减少, 那么是会影响比特币网络的可靠性和可用性的。当矿工节点或者说矿池打包出一个区块后, 就需要向网络广播, 这个“向网络广播”, 实际



上就是向网络中的全节点广播。全节点接收到区块后,会对区块进行验证,验证通过后则会写入到本地的区块链账本数据中,并且会继续传播这个区块,以使得其他节点也能获得相同的区块数据。每一个节点接收到新的区块数据后都会经过一个验证后上链。

#### 4. 矿池间的数据传播

挖矿程序在进行运算的时候,实际上是对新区块区块头的组成参数进行不断的哈希,直到找到符合难度目标值的那个随机数。因此在矿池间进行挖矿计算的时候,是不需要将整个新区块数据进行传播的,而只需要传播区块头即可,这些是挖矿计算程序的必备参数。计算对于连接矿池的矿工来说,也仅需要知道新区块的部分信息,也就是区块头后就可以开始进行挖矿计算了,因此不用接收区块的完整数据,从而可以提高挖矿的效率。

### 7.3 比特币的发行机制

#### 7.3.1 总量上限 2100 万的实现

##### 1. 区块新币奖励及其减半

比特币的发行是通过挖矿奖励进行的,每当一个区块被挖出就会根据奖励规则奖励一定数量的比特币。比特币中的新币奖励是递减的。在一开始的时候,每个区块的新币奖励是 50 个比特币,然后每隔 210 000 个区块减半一次,按照比特币的平均出块时间 10 分钟来计算,也就是约 4 年时间进行一次奖励减半。如果我们把 210 000 个区块的递减周期看成一个整体的话,那么可以发现,这个奖励实际上是一个递减等比数列,如 50, 25, 12.5... 直到最后奖励不足 1 聪。这样的等分会进行 64 次,我们可以写个程序或者用等比数列的求和公式计算出比特币的发行总量,大约是 2100 万枚。

为什么说是大约呢?因为我们可以看到这里面的数据随着等比递减,并不是整数,而是小数,实际上最终计算出的值是 20 999 999.9769 这样的近似数。

##### 2. 交易手续费计算及分配

比特币在转账交易的过程中,是要支付手续费的。这是因为作为一个点对点的公共分布式系统,需要依靠矿工打包区块以进行记账,需要给这样的行为一个激励。在之前的阐述中提到过新币的奖励,这是一种激励方式,除了新币奖励外还有手续费的激励。

比特币交易的手续费是可以设置的,打包区块的矿工可以设置自己接受的手续费额度。目前每笔交易的手续费默认为 0.000 01BTC/KB,也就是每条交易占用的数据量在 1000 字节内的话默认就是 0.000 01 个 BTC 的手续费。我们来看一下图 7-4 所示的比特币核心客户端所带钱包功能的界面:

如图 7-4 所示,在发起一笔转账交易时,矿工是可以自行设定的,也可以仅设定为默认的必要费用。当然,比特币对手续费的设置并没有强制要求,即便是设置为 0 也是可以

的，只不过这样会降低交易事务的优先级，可能会有较长的确认等待时间。



图 7-4 核心客户端交易费用设置

### 3. 区块交易手续费将超越区块新币奖励

通过上述的介绍我们知道，比特币中是设计了一组简单的经济模型的，包含新币的发行以及交易手续费的设计。对于矿工来说，这两者也是主要的激励来源。然而随着时间的推移，新币的奖励是逐渐减半的，到最后新币奖励就停止了。新币奖励停止后，矿工的收入主要就是手续费了。失去了可观的新币奖励后，矿工必然会提高手续费的设置，低手续费的交易将会很难有效地被打包进区块。实际上，就之前被打包的区块数据来看，已经出现了区块中的手续费比新币奖励更高的情况。比如我们查看第 494 045 区块可以看到，区块中总计手续费高达 13.400 622 21BTC，而区块中的新币奖励才 12.5BTC。未来当区块奖励停止后，手续费很有可能会更高，矿工节点为了维持运转而付出的成本如果小于收益的话，就会停止运行。这对于比特币网络未来的持续运行也是一个挑战。

## 7.3.2 打包区块的原理

通过比特币客户端发送交易事务后，会广播到网络中等待被打包确认。在被打包到区块之前，会保存在一个叫做交易池的内存区域中。矿工会从交易池中按照一定的规则以及优先级获得相应的交易事务，构造出一个区块。这个过程就是比特币的区块打包过程，其中有哈希函数的应用，也有工作量证明机制的应用。接下来我们就来逐一介绍。

### 1. 哈希函数

前面我们讲过，哈希（Hash）函数也叫散列函数，它的作用如下：将任意长度的字符串通过特定的算法过程转换为固定长度的字符串输出，也就是哈希值。无论目标字符串有多长，输出的值都是一个固定的长度，并且对于不同的字符串，输出的哈希值也是不同的，因此哈希值相当于一个身份证号或者是指纹数据。具体来说，哈希函数具有如下特点。

1) 单向性。通过对字符串进行哈希计算可以得到哈希值，但是通过哈希值通常是很困难，或者说是无法得到原字符串的，这个过程并不是可逆的。也因此哈希函数并不能用来作为加解密算法，只能用来生成某段数据的特征码。

2) 唯一性。只要字符串不同，则经过同一个哈希函数计算出的哈希值肯定是不一样的，



这也是很多场合可以用数据的哈希值来表示原数据的原因。当然，在海量数据之下，可能会存在一定的碰撞概率，因为哈希值是定长的而原始数据是不定长的，只要基数够大，就存在一定的碰撞概率，即使是很低的概率。这种情况下，会使用多种不同的哈希函数来计算，将多个不同的值联合起来，降低碰撞的概率，这也是布隆过滤器的原理。

3) 效率高。对一段字符串计算哈希值，这个过程通常是很快的，而很多加密算法在对字符串进行加密的过程效率却不一定很高，比如公开密钥算法中的签名计算，如果要对各种不定长的数据进行频繁的签名计算，效率不高，此时如果首先对数据进行哈希计算，生成较短的、定长的哈希值，然后再对哈希值进行签名，效率就高很多。

哈希函数的种类是非常多的。比特币中主要使用的有 SHA-256 以及 RIPEMD160，尤其是 SHA-256，用于计算区块头的哈希值，也用于计算交易事务的哈希值，还用于实现挖矿算法。

## 2. PoW

通常所说的比特币中的挖矿就是一种工作量证明算法。顾名思义，工作量证明就是通过某种方法证明自己完成了符合要求的工作量。那么具体是什么意思呢？

我们知道比特币是一种点对点的分布式网络结构，并且比特币的客户端是运行在互联网之上的，网络状况复杂，且不能保障节点运行的稳定性，任何一个节点都可以随时加入比特币网络，也可以随时退出网络。在这种情况下，依靠什么样的机制去保持每个节点之间的数据一致是一个难题。根据 FLP 定理，在异步通信场景（互联网就是典型的异步通信场景）中，即使只有一个进程失败，也没有任何可靠的算法能保证所有进程能够达到一致性。比特币网络提供的是什么样的解决方案呢？那就是 PoW。

在比特币的设计中，区块打包是一种需要竞争抢夺的权利。在每个区块打包之前，都有一个相应的难度目标值，矿工节点需要通过一种很耗费算力的计算来获得一个符合难度要求的结果；谁先获得结果，谁就获得打包权，获得打包权的矿工可以将网络中的交易事务打包成区块并广播出去；其他节点接收到新区块后就会进行验证，验证通过后写入自己本地的区块链账本中；如果是挖矿节点接收到新区块，就会停止当前区块的竞争计算，转入下一个区块的难度竞争。

这个竞争的计算过程，是对区块头的参数进行连续的哈希计算，通过不断地调整参数中的随机数，使得计算出的结果小于难度目标值。这类似于掷骰子的游戏：大家约定一个 256 位的数字，然后每个人掷 256 次骰子得到一个 256 位的数，这个得出的数字必须比约定的数字要小，而且前面 N 位还必须都是 0。可以想象得到，为了得出这么一个结果，就只有不断地去试，这是个很累人的活。比特币中采取的就是类似的原理，通过这种方式筛选出交易数据的打包者，其他节点以打包者生产出的区块数据为准，这样每个节点都间隔地接收一个个新区块，构成一个区块链的数据结构。

我们来具体看下比特币中的工作量证明是怎样一个过程。我们先来看一下比特币的区块头的机构，如图 7-5 所示。

比特币中的工作量证明算法，也就是挖矿过程，跟区块头中的这几个数据息息相关。通过对这些数据的计算，得出的结果要小于难度目标值才行。那么矿工需要去进行怎样的计算呢？我们来看一下比特币中挖矿计算的公式：

SHA-256(

SHA-256( 区块版本 + 前一个区块的哈希 + Merkle 根 + 时间戳 +  
当前难度 + 随机数 ) < 难度目标

)

通过上述公式我们可以看出，在所有参与计算的参数中，可以调节的主要是 Merkle 根时间戳和随机数，需要去不断地尝试变更参数来计算出符合要求的值。显然这个过程要频繁地进行哈希计算，要想抢先算出来，是很考验算力的。

PoW 算法其实就是提供了一种共识验证的方法。在这种机制下，大家可以相对公平地去竞争，依靠的就是设备的算力，除此以外并没有什么可以讨巧的地方。其他节点对于生成的新区块可以很容易去验证计算出的结果是否符合难度值，只要按照规则公式做同样的哈希计算，然后比较结果即可。之前我们介绍哈希函数的时候提到过，哈希计算过程的效率是很高的。这种机制还有一个好处，那就是验证者在验证过程中并不需要去与挖矿节点交互，整个验证的过程就是比较一下难度值是否符合要求即可，即整个过程是可以独立进行的。

|          |
|----------|
| 区块版本号    |
| 前一个区块的哈希 |
| 梅克尔根     |
| 时间戳      |
| 当前难度     |
| 随机数      |

图 7-5 比特币区块头

### 3. 打包区块的结构过程

现在我们来具体了解一下比特币中的区块打包过程。首先我们来看一下大概的步骤。从交易事务的签发到打包，历经了如下的过程：

- 1) 接收并验证交易事务，并将交易事务放入交易池中；
- 2) 将交易池中的每笔交易按照规则分配优先级；
- 3) 从交易池中取出交易事务构建区块；
- 4) 进行挖矿算法的计算，抢夺打包记账权；
- 5) 挖矿成功后组装完整的区块并广播给相邻节点；
- 6) 其他节点接收到新区块后进行验证，并加入本地的区块链账本；
- 7) 其他挖矿节点接收到新区块后会放弃当前区块的挖矿计算，立即进入下一区块的挖矿计算；
- 8) 新区块不断地在网络中传播。

注意，在整个挖矿计算过程中，能够计算出符合条件值的节点不一定只有一个。那么，如果有多个矿工节点都计算成功，都打包了区块在网络中广播，那会发生什么呢？对于接收新区块的节点来说，都是符合条件的，就都会接收，这时候区块链账本会产生临时分叉，也就是说从结构来看，比特币的区块链结构会像树木的枝杈一样产生分叉。而对于比特币



节点来说,在不断有新的区块进来的过程中,总是会选择最长的那条链作为主链。

### 7.3.3 矿池与矿工的关系

简单地说,矿工是执行挖矿算法的程序,而矿池是用来组织大批量的矿工一起来挖矿的程序。矿池与矿工之间是通过一组专用协议来交换数据的。

#### 1. 矿池的分布

目前比特币已经基本不存在个体挖矿了。由于挖矿难度已经到了相当大的地步,个体矿工挖到矿的概率极其渺茫。矿池出现后,比特币的整个算力网络就逐渐由若干大的矿池分配和承担。我们来看一下比特币矿池的分布。图 7-6 的数据是从 <http://qukuai.com/pools> 获得的即时统计。由于数据是会发生变化,因此大家可以自行获取最新的统计结果。

| 矿池             | 区块数       |               |
|----------------|-----------|---------------|
| <b>BTC.TOP</b> | <b>29</b> | <b>19.59%</b> |
| BTC.com        | 25        | 16.89%        |
| AntPool        | 23        | 15.54%        |
| SlushPool      | 16        | 10.81%        |
| ViaBTC         | 15        | 10.14%        |
| BitFury        | 5         | 3.38%         |
| GBMiners       | 4         | 2.70%         |
| BTCC           | 2         | 1.35%         |
| BitClub        | 2         | 1.35%         |
| 1Hash          | 1         | 0.68%         |
| Bitcoin.com    | 1         | 0.68%         |
| Kano CK        | 1         | 0.68%         |
| unknown        | 1         | 0.68%         |
| BATPOOL        | 1         | 0.68%         |
| 未知             | 22        | 14.86%        |

图 7-6 矿池分布图

从图 7-6 可知,大的矿池占据的算力是相当大的。在矿池背后,连接着的是成百上千万的矿机,这些矿池构成的算力便是维持整个比特币网络的基石。

#### 2. 矿工为矿池提供算力

如上所述,矿工是真正的挖矿设备,矿池只是一个服务者或者说是组织者。矿池相当于一个网络服务,为矿工分配计算任务,获得矿工的计算结果以及分配收益给矿工。目前比特币矿工与矿池之间主要是通过 Stratum 协议进行通信的。Stratum 协议是 JSON 格式。矿机启动后,与矿池连接并且订阅工作,矿池会定期分配任务给矿机,矿机计算出需要的结果后提交给矿池,期间如果比特币难度有调整,矿池还会将新的难度值下发给矿机。

加入矿池的矿机越多,矿池的整体算力就越强,这跟云计算或者集群计算是类似的。矿池一般都是对外开放注册的,任何人都可以将自己的矿机注册到某一个矿池,作为矿池算力的一个组成。

### 3. 矿池为矿工提供简化计算数据

如果不通过矿池来挖矿，而是通常的个体挖矿或者说是标准的挖矿过程，那就需要矿工自己准备所有的计算所需参数，从获取交易事务开始，计算出 Merkle 根以及准备好其余的参数，然后开始不断地进行挖矿计算。整个挖矿过程，从准备数据、计算到验证都要自己完成。而连接矿池的矿工并不需要直接与全节点账本数据连接，而可以通过协议直接从矿池服务程序获得挖矿的参数来挖矿，矿池充当服务者的角色。

### 4. 矿业最新 ASICBoost 专利技术

ASIC 就是集成电路矿机，是目前比特币挖矿的主流设备。ASICBoost 是一种提升矿机挖矿效率的技术。我们知道比特币挖矿的过程无非就是对区块头的组成参数进行双哈希计算。在几个参数中，上一个区块的哈希值是固定的，当前难度也是固定的，版本号更是固定的，能够动态调整的就只有随机数、Merkle 根和时间戳了。

对于随机数这个选项，其大小只有 4 个字节，也就是说搜索空间为  $0 \sim 2^{32}$ ，空间大小为 4 个 GB。当随机数的搜索空间不能计算出需要的结果时，就要组合时间戳和 Merkle 根的改动了。其中时间戳的更改范围是有限的，那么主要就靠 Merkle 根的改动了。要改动 Merkle 根，就要重新组合交易事务或者更改 Coinbase 交易数据。好在 Merkle 根的字节数是固定的，有 32 字节，可以提供足够大的搜索空间。

这几个参与挖矿计算的参数，虽然有 3 个是可以变动的，但是参数的顺序是不能变的。通过上述的挖矿计算公式我们知道拼接的顺序如下：

区块版本 + 前一个区块的哈希 + Merkle 根 + 时间戳 + 当前难度 + 随机数

现在的问题是，如果有一个方法可以减少搜索空间而更快地得到结果，那就能提高挖矿效率了。

AsicBoost 就是一种提高矿机挖矿效率的技术。它可以减少矿机电路的运行损耗，从而提高效率的技术，具体来说就是优化挖矿算法中的 SHA-256 计算过程。SHA-256 算法在处理数据的时候，会按照 64 字节一组去处理。比特币的区块头组成是 80 字节，按照 64 字节一组，显然第二组是不够的，因此需要扩展，扩展后参与计算的区块头字符串就是 128 字节。对于分出的 64 字节数据段，SHA-256 还会按 4 字节一组拆分并且进行 64 轮计算。那么，我们再来看区块头中的数据被这样拆分后会怎样。

如图 7-7 所示，前面 64 字节构成一个数据块，后面 64 字节也构成一个数据块。我们发现，Merkle 根的部分被拆分成了前段和后段，分别处于前后两个 64 字节的数据块中。通过观察这样一个结构可以发现，在进行 SHA-256 计算的时候，前面 64 字节的数据块倘若保持不变的话，那么在运算的时候只需要进行后面 64 字节的计算，然后再连接起来即可。

ASICBoot 就是这种思路，不过 ASICBoot 是将后面 64 字节固定住。因为实际计算的时候，时间戳基本是稳定的，而随机数的范围就只有  $2^{32}$ ，选定一个随机数后，如果能够选出后面 4 个字节共同的 Merkle 根，则主要的参数变更就在 Merkle 根的 28 字节这部分。



加之搜索空间也足够大，只要碰撞出后面 4 个字节共同的 Merkle 根就行了。这种方式可以减少电路的运算，对于 ASIC 矿机来说，可以去掉一些不必要的电路，提高碰撞的效率。究其本质，实际上就是对 SHA-256 计算的电路优化。

| 区块头  |         |       |      |      |      |      |       |
|------|---------|-------|------|------|------|------|-------|
| 版本号  | 前一个区块哈希 | 梅克尔根  |      | 时间戳  | 难度值  | 随机数  | 补足字节  |
|      |         | 前段    | 后段   |      |      |      |       |
| 4 字节 | 32 字节   | 28 字节 | 4 字节 | 4 字节 | 4 字节 | 4 字节 | 48 字节 |

图 7-7 区块头字节拆分

## 7.4 比特币的账号系统

### 7.4.1 私钥与公钥

比特币中账号系统的设计完全不同于通常软件设计中的用户账号模型，而是使用了公开密钥算法的私钥和公钥来表示，具体来说使用了椭圆曲线密码算法实现的数字签名算法。比特币使用了一对密钥来表示账户：通过私钥可以签发交易、表示账户地址；在比特币中，账户地址是通过对公钥进行一系列步骤转换得到的。私钥与公钥是成对匹配的，用私钥签名的数据只能由公钥来解密，反之使用公钥加密的数据只能用私钥来解密。私钥是必须妥善保管并且保密的，公钥可以公开。通过公/私钥的组合可以实现很多安全方案，比如用私钥签发一笔数据然后发出，其他人使用公开公钥来解密，如果能成功就证明数据确实是合法的，反之就说明数据是伪造的。

#### 1. 私钥：由 256 个 1 或 0 形成的随机数

私钥实际上就是一个随机数，长度为 256 位。随机生成一个私钥后，可以根据椭圆曲线算法（比特币中使用的是 secp256K1 算法）来生成公钥。私钥是使用了 SHA-256 算法生成的。256 位的随机哈希计算可以表示的范围很广泛，取值范围在  $0 \sim 2^{256}-1$  之间，不过附加 secp256K1 曲线的取值约束后，实际可以的范围是从 0x01 到 0xFFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF BAAE DCE6 AF48 A03B BFD2 5E8C D036 4140 之间。通常我们在钱包中看到的私钥数据格式都是经过了编码转换的，因为 256 位长度的字符串实在是不方便保管携带，容易拷贝遗漏，比如钱包会支持 WIF (Wallet Import Format) 格式，这是经过了 Base58 编码后的格式。

也由于私钥其实就是一个 256 位长度的二进制数字，因此我们甚至可以自己用掷骰子的方式来生成一个私钥，比如约定点数小于等于 3 的表示 0，点数大于等于 4 的表示 1，连续掷 256 次也能得到一个私钥。

有人可能担心，万一自己获得的私钥与别人的一样怎么办？那是不是就是破解了别人

的私钥了？正好这个私钥下还有比特币的话，那它不是赚了？实际上这里面是有一个概率问题的。256 位的长度，总共可以表示 2 的 256 次方个数字，几乎接近人类可观测到的宇宙中的原子总数，在这么大的一个范围内要想正好命中一个别人的私钥，这个概率小得接近于零。

## 2. 带校验的 Base58 编码

前面提到了比特币的账户地址是通过公钥转换而来的，在这个转换的过程中就使用到了 Base58 编码。我们先来看一下什么是 Base58 编码。顾名思义，Base58 就是包含有 58 个组成符号。Base58 编码是由 58 个字母和数字组成的，如下所示：

123456789ABCDEFGHJKLMNPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz

在 Base58 的编码中，不包括以下字符：

- ☐ 数字 0
- ☐ 大写字母 O
- ☐ 大写字母 I
- ☐ 小写字母 l
- ☐ + 与 /

这主要是为了防止混淆：数字 0 和小写字母 o 很容易混淆，小写 l 与大写 I 也很容易混淆，所以排除在外。比特币中使用的 Base58 其实还作了改进，增加了校验功能，因此，比特币中使用的其实是 Base58Check 编码。具体是什么意思呢？

在二进制数据的传输过程中，为了防止数据传输的错误，保护数据安全，通常会加一个校验码，通过校验码的配合可以发现数据是否被破坏或者是否在发送时输入错误了。Base58Check 就是 Base58 编码加上校验码。在下一节我们介绍比特币地址生成过程中，会看到 Base58Check 会做怎样的处理。

## 3. 从私钥计算公钥和地址的过程

通过上述的介绍，我们已经了解到比特币中的账户是通过私钥和公钥来表示的，而公钥是通过私钥生成的，地址又是通过公钥转换而来的。那么，这其中具体是怎么生成的呢？我们来看一下示意图，如图 7-8 所示。

如图 7-8 所示，私钥通过 secp256k1 算法计算出公钥，接下来就是对公钥进行一系列的步骤转换来生成地址。图中的 RIPEMD160 也是一种哈希算法。公钥经过 SHA-256 和 RIPEMD160 两道哈希计算后，得到了一个公钥哈希。在公钥哈希的前面放上一个 0x00 的版本号，版本号与公钥哈希连起来后再次进行两次 SHA-256 哈希计算，得出的值取出其中的前 4 个字节作为校验值。从图中我们可以看到，比特币中将版本号 0x00 和公钥哈希以及校验值联合起来进行了 Base58 编码计算，这种带上了校验值的 Base58 计算就是 Base58Check 计算。通过这一系列的步骤最终生成了比特币地址。



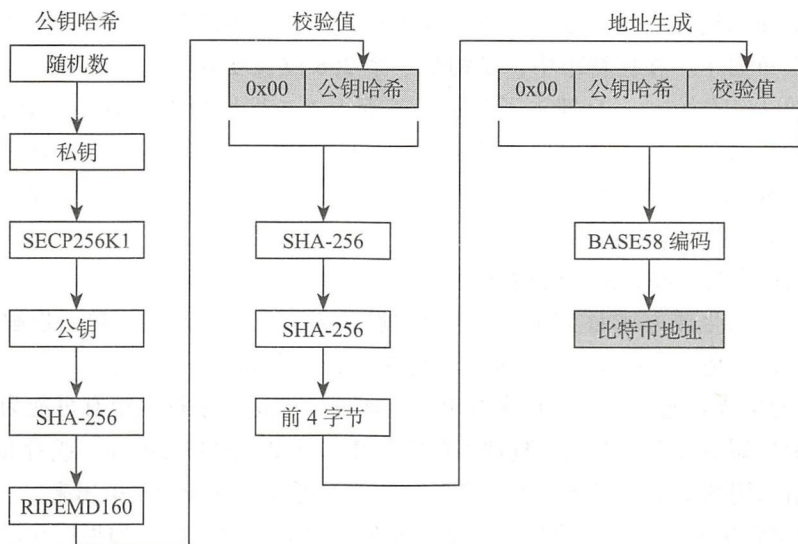


图 7-8 比特币地址生成过程

## 7.4.2 签名信息与校验签名

### (1) 用私钥签名信息

我们知道，在比特币中是通过私钥和公钥来表示一个账户的。私钥可以用来进行签名计算。所谓签名其实就是加密，只不过通常在公开密钥算法中，习惯称之为签名。使用私钥可以签名任何消息字符串。注意，使用私钥签名消息时，通常并不是直接对消息进行签名，而是对消息的哈希值进行签名。

### (2) 校验签名信息

对于私钥签名的信息，需要进行校验以确定该签名是否是地址持有人创建的，在校验时根据消息签名计算出 pubkeyID，然后生成比特币地址。如果生成的比特币地址与公开的地址是一致的，则表示验证通过，否则校验失败。

### (3) 零知识证明：不需展示私钥证明拥有私钥

零知识证明是一种数据拥有证明技术。所谓的零知识，就是说在证明过程中不需要暴露自己的私密数据而可以向别人证明自己拥有私密数据。比如说我可以通过打开保险箱来证明我拥有打开保险箱的密码，而同时我不需要向别人暴露保险箱密码。实际上，上述介绍的私钥签名就是一种零知识证明技术，因为我们在整个签名的验证过程中并不需要知道私钥是什么。

## 7.4.3 脑口令

### (1) 脑口令简单原理

上述我们了解到，比特币地址的生成是先生成私钥，私钥再计算出公钥，公钥再转换

为地址。这里面私钥是最为关键的，只要有了私钥，其他的都是可以生成出来的，因此私钥的获得就很重要了。在比特币中，私钥是一个 256 位长度的随机二进制串，显然这么长的串我们很难去记住它。一旦遗失了私钥，就等于永久性地失去了对某一地址的所有权。因此脑口令的做法就产生了，用户通过输入一串自行指定的字符串，然后对这个字符串进行一次 SHA-256 计算，生成的 256 位的哈希值就作为私钥。显然在这个过程中，只要记住那个自己指定的字符串就能找回私钥。

### （2）对脑口令与脑钱包的安全质疑

通过脑口令的生成方式而实现的钱包就叫**脑钱包**。虽然做法很简单，也解决了难以记住私钥的问题，可是安全性却受到质疑。具体来说，它有如下一些问题。

1) 私钥的生成是通过一个用户字符串来生成的，而这个字符串很有可能为了方便而指定了一个很短的短语或者一个很有规律性的字符串，比如 123456abcde，或者 helloworld 之类，这将会给私钥的安全性带来很大的隐患，其他人可以通过猜测模拟出来。

2) 用户字符串的保存是一个问题，而且也同样是容易遗忘的，如果记在纸上，不但不安全，而且也失去了脑钱包或者脑口令的初始意义了。

3) 增加了用户的使用麻烦，实际上用户根本就无法去决定一个怎样的字符串才算是安全的，这会给使用带来困扰，而且一不小心就会生成一个不安全的私钥，还不如让程序来随机生成。

当然，脑口令和脑钱包的使用毕竟能带来方便，在这个基础上如果再增加些安全措施，还是可以推荐使用的。

### （3）“左脑 + 右脑”的安全脑口令

所谓的“左脑 + 右脑”，其实是一个形象的说法。我们知道，大脑分为左脑和右脑，左脑比较擅长逻辑推理、数学分析，右脑更擅长图像识别、音乐艺术等活动。那么，这些与脑口令有什么关系呢？脑口令实际上就是一个思路，它的过程大概是这样的：

1) 准备一段自定义的字符串，就跟上述生成脑口令的做法一样，至于这个字符串要多长、复杂点还是简单点，可以自行选择，总之这个字符串定义后要能记住；

2) 根据字符串生成私钥，过程跟普通的脑口令一样；

3) 对生成的私钥进行某个位置的修改，比如修改某一位的字母或者数字等。

通过上述方式生成的私钥，就要记住原始字符串，以及对生成私钥后的几个位置修改，相当于左脑记住字符串，右脑记住位置修改，这样可以生成更加安全的脑口令。

### （4）脑分裂口令钱包工具

脑分裂口令，顾名思义，就是将一个私钥拆分若干份，只要集齐就可以获得真正的私钥。这相当于将私钥分开来藏在不同的地方，类似于古代的虎符管理，分成两份或者几份，大家一起拿出来才能有效。从这一点来看，脑分裂口令的原理与多重签名地址有异曲同工之妙，其目的无非是为了能够方便找到私钥的同时还能妥善地安全保管。



## 7.4.4 荣耀地址与批量地址

### 1. 如何大批量生成币地址

我们知道,要生成不同的比特币地址,实际上就是要生成不同的私钥,只要能批量生成私钥,自然就能得到批量的地址了。对于私钥,通过上述的介绍,我们知道就是 256 位长度的随机比特串,那么我们就有多种方法来批量生成比特币地址了,如下所示。

1) 准备好大量的已知字符串,比如诗词或者字母加上数字的排序组合,然后用脑口令的方式一次性生成,保管好字符串表;

2) 指定一个原始字符串,在这个字符串的末尾设置为若干位数字,只要循环不断地对尾部的数字加 1,就可通过工具生成地址;

3) 分层生成批量地址,先生成一个私钥,然后通过这个根私钥再生成子私钥,以此类推,只要生成子私钥的规则是固定的,则只要记住根私钥就行了。

批量生成地址的方式,本质上就是提供给地址生成程序一个可以快速处理的规则,我们要注意,只是批量生成是不够的,还要同时能有效地管理这些批量地址才行。

### 2. 筛选生成荣耀地址

荣耀地址有点类似于 QQ 靓号或者手机靓号,是指在生成的地址字符串中包含一部分可读信息,比如我们看一个地址 1Bitcoin74Ysfsf88sVN5nAHfgr9Fjbs22,在这个地址中包含有一个 Bitcoin 的可读信息,这个地址就是一个荣耀地址。

荣耀地址的生成需要对批量生成的私钥来进行测试,直到找到符合要求的地址为止。通常使用单机的地址生成器来匹配所需的荣耀地址,过程可能会很漫长。除了单机计算匹配外,还可以通过矿池服务来生成。通过矿池可以将匹配计算委托给更多的机器,能够比较快地找到符合要求的地址。

需要注意的是,荣耀地址跟普通地址在技术意义上是一样的,并没有什么差别,只是其地址字符串中包含了可读字符串而已。

### 3. 分层生成批量地址

在比特币中,有一类钱包叫做分层确定性钱包,所提供的技术就是分层生成批量地址。实际上原理很简单,还是回到一开始的概念,要生成地址就要先生成不同的私钥,有不同的私钥就有不同的公钥,然后就能有不同的地址。所以,只要能快速地生成不同的私钥就可以了。

我们可以先提供一个初始的私钥,然后通过一个固定的算法来生成一棵私钥树,形状如图 7-9 所示。

生成一棵私钥树后,可以快速通过主私钥和子私钥来生成地址,只要保持主私钥和子私钥以及子私钥与它的子私钥之间的生成关系是固定的,那么整个私钥树的保存实际上只要保存好一个主私钥就可以了。而在有些场合,为了更进一步地提高安全性,子私钥是可以不用保留下来的,因为私钥毕竟不能公开,而可以生成如图 7-10 所示的树结构。

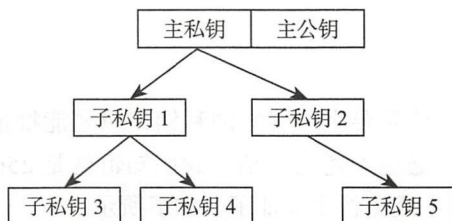


图 7-9 分层私钥生成

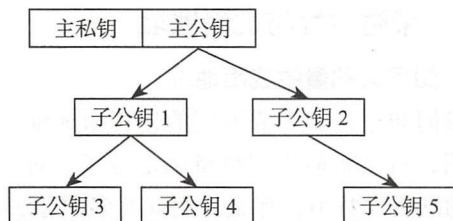


图 7-10 分层公钥生成

如图 7-10 所示，公钥都是可以放心地公开的，只要保管好主私钥就可以了。在整个做法中，需要注意两点：

- 1) 生成一个足够随机的主私钥，不能让别人可以简单地就碰撞出来；
- 2) 主私钥与子私钥之间设定一个固定的生成规则，比如在主私钥的基础上不断地递增一个数字或者其他某个规则。

#### 7.4.5 多重签名地址

如果大家注意观察的话，会发现在通过比特币钱包生成地址的时候，通常地址的第一位都是数字 1，这是比特币中使用最广泛的地址格式。基于这样的地址格式实现的交易脚本，称为是 P2PKH，也就是 Pay-to-Public-Key-Hash，翻译过来是“支付到公钥哈希”的意思。我们知道，比特币的地址是通过公钥经过一系列的步骤（主要是哈希计算）转换而来的，因此称之为支付到公钥哈希。然而在比特币的发展过程中还出现了一类地址，叫“多重签名地址”。对于这种地址的交易，底层的交易脚本需要多个私钥签名来证明所有权，才可以签发交易，类似于传统的会审。在多重签名实现的过程中，通常会有多个密钥比如 N 个加入，而签发交易的时候会需要至少 M 个签名才有效，表示形式为“M-N 签名”，比如 1-2 签名表示在多重地址中，总共有两个私钥联合组成，而实际签发时，至少有其中一个签名就可以了。与多重签名地址相关的交易脚本通常是 P2SH，也就是 Pay-to-Script-Hash，翻译过来为“支付到脚本哈希”，也就是说不是支付到某一个公钥所有者，而是支付到一个脚本程序，通过脚本程序具体定义了上述的 M-N 的验证过程。当然，并不是说只要是 P2SH 的支付脚本就是多重签名地址的方式，只不过通常使用 P2SH 的方式来实现这种类型的交易而已，P2SH 还可以实现其他类型的脚本交易。

##### 1. 理解比特币的脚本

在比特币中的转账交易过程中，我们通常都会说使用私钥签发一笔交易，然后广播到网络中进行验证并被打包进区块。在这个过程中，具体是通过什么样的方式来进行验证处理的呢？在底层实现上，主要是通过一组指令构成的脚本来实现的，而脚本是一笔交易事务的组成部分。我们先来看一下比特币中交易事务的构成。为了看清楚原理，我们截取比特币源码中对交易事务的定义源码：

```
std::vector<CTxIn> vin;
```



```
std::vector<CTxOut> vout;
int32_t nVersion;
uint32_t nLockTime;
```

源码定义中的 `nVersion` 是指版本号, `nLockTime` 是指锁定时间, 这两个字段我们暂时忽略。其余两个就是交易的主要组成, 一个是 `vin`, 一个是 `vout`。通过前面章节的学习, 我们已经知道了比特币中的交易事务是由输入和输出组成的, 这里的 `vin` 部分就是指输入, `vout` 就是指输出部分。我们再来看看 `vin` 和 `vout` 中都定义了什么。

```
class CTxIn
{
public:
    COutPoint prevout;
    CScript scriptSig;
    uint32_t nSequence;
```

上述是 `vin` 的结构定义, 其中 `prevout` 是指向之前的交易输出, 也就是其他人转账给自己的交易输出, 这部分原理我们在之前的章节中已经阐述过了, 这里不再赘述。`nSequence` 是指定的之前的交易输出的序号, 因为之前的交易输出不一定只有一个, 所以需要指定一个序号, 表示是哪一笔。其中的 `scriptSig` 就是我们所说的输入脚本, 也就是交易的签名程序, 在这里我们可以看到, 签名实际上指的是签发之前的交易输出, 因此输入脚本也叫解锁脚本, 解锁了之前接收到的交易输出, 表示要进行新的转账了。

我们再来看 `vout` 的定义, 如下:

```
class CTxOut
{
public:
    CAmount nValue;
    CScript scriptPubKey;
```

可以看到, `vout` 的结构很简单, `nValue` 就是转账的金额, `scriptPubKey` 就是输出脚本。输出脚本是用来表示接收者的, 通过输出脚本可以锁定这笔交易, 只有接收者使用自己的私钥才能来签名使用, 因此输出脚本也叫锁定脚本。

我们可以看到, 在比特币的交易事务中, 具体的执行实际上主要是依靠输入和输出脚本, 在比特币的脚本实现中, 并不是通常的像 JavaScript/Java/C++ 这样的语言实现的图灵完备的程序, 而是使用的逆波兰表达式, 是一种栈式结构。当输入脚本和输出脚本结合在一起的时候, 就能构成一个完整的指令程序, 类似虎符一般, 碰到一起就能生效了。

我们来看一个 P2PKH 格式的交易脚本组成:

```
OP_DUP OP_HASH160 <PubkeyHash> OP_EQUALVERIFY OP_CHECKSIG
```

上述指令是一段输出脚本, 其中 `OP` 开头的都是指令。`PubkeyHash` 是指公钥哈希, 发起交易者要使用自己的私钥解锁之前的可用交易输出, 其脚本指令如下:

```
<Sig> <PubKey> OP_DUP OP_HASH160 <PubkeyHash> OP_EQUALVERIFY OP_CHECKSIG
```

可以发现在输出脚本之前加上了一段签名数据，组成了完整的执行指令。上述的指令实际上就表示一笔交易的签发执行过程。假设 Alice 发起交易给 Bob，则 Alice 要做的事就是：首先解锁之前的交易输出，然后构造对 Bob 的输出脚本放到本次交易事务中。我们来看下 Alice 是怎么解锁之前的交易输出的。假设 Alice 使用的是之前 Lily 发送给自己的交易输出。它的执行过程大概如图 7-11 所示。

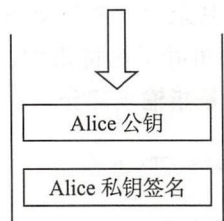


图 7-11 压入 Alice 私钥签名与公钥

这里的私钥签名就是<Sig>部分，公钥就是<PubKey>部分，这两个输入先后压入一个栈结构中，然后遇到一个 OP\_DUP 指令，这是一个栈顶拷贝指令，执行后变成如图 7-12 所示的结构。

可以看到，栈顶复制了一个 Alice 公钥，我们继续执行指令。后面的指令是 OP\_HASH160，通过这个指令就完成了将栈顶的 Alice 公钥转换为地址的过程，执行后数据结构如图 7-13 所示。

接下来将<PubkeyHash>部分压入到栈结构中。PubkeyHash 是指 Lily 发送给 Alice 的交易输出部分的公钥哈希。我们可以发现，从这一步开始，要做的事情其实就是要来验证 Alice 对之前的交易输出是否具有使用权。数据压入栈中后，数据结构变为如图 7-14 所示。

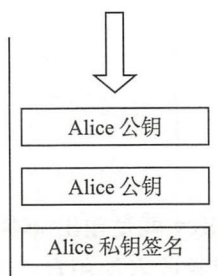


图 7-12 复制 Alice 公钥

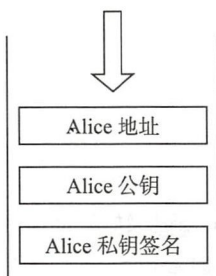


图 7-13 栈顶公钥转换为地址



图 7-14 压入前置交易中的公钥哈希

到了这一步，就可以执行后面的 OP\_EQUALVERIFY 指令来校验两个 Alice 地址是否一致了。如果不一致则校验失败，如果一致则删除栈中这两个 Alice 地址。删除后继续执行最后一个 OP\_CHECKSIG 指令，校验一下 Alice 的公钥和私钥签名是否匹配，证明这笔交易是不是 Alice 签发的。验证通过则这笔交易就可以签发出去了。

在 Alice 给 Bob 的转账交易过程中，事务的输出脚本部分就是构建的针对 Bob 的输出验证脚本，当 Bob 以后想要使用这笔针对自己的交易输出时，也需要使用如上的步骤来验证。而对于支持多重签名地址交易的 P2SH 脚本指令，其实主要就是支持了在输出 / 脚本中可以包含另外的脚本哈希，这样就不只是转账给一个单独的公钥哈希了，验证方式也更换





为 OP\_CHECKMULTISIG。在多重签名的验证场合支持的是 M-N 的验证，因此脚本组会多一些验证所需的数据。究其本质，还是类似的，都是在这样一个栈结构中进行判断验证。

## 2. 生成以 3 开头的币地址

多重签名地址就是以 3 开头的，我们可以在比特币客户端的命令控制台进行生成。假设我们要生成一个 2-3 的地址，首先要生成 3 个地址。通过 `getnewaddress` 可以生成，如图 7-15 所示。

生成了 3 个地址后，我们要来生成多重签名地址。显然既然是多重签名，那就要包含进去这 3 个地址，同时我们要的是 2-3 的签名模式，因此也要指定这个参数，使用的创建命令如下：

```
createmultisig 2
['18Bur9gfCxuc8q2YtZ8VU4bD3AkX8M9ecN',
'1KugGHN1BAiuJ5EKmPJp6RRpfawCrfJwrP',
'1ChKqFq4vA5jBG247vezgZdqBGXcRwq2XF']
```

```
getnewaddress
18Bur9gfCxuc8q2YtZ8VU4bD3AkX8M9ecN
getnewaddress
1KugGHN1BAiuJ5EKmPJp6RRpfawCrfJwrP
getnewaddress
1ChKqFq4vA5jBG247vezgZdqBGXcRwq2XF
```

图 7-15 比特币地址生成

我们使用了 `createmultisig` 命令，其中的参数 2 就是指我们需要同时 2 个私钥签名才能签发一笔交易，后面就是 3 个创建的地址了。命令执行成功后，输出如下：

```
{
  "address": "3NapKrz8mEDGT9enQdrftU1PRZ35L9E6mY",
  "redeemScript":
  "522102d9e3b6248770334747947f78596d5081c3f167aeb171bfbcb86e8d
567671912b921021e7f3fbee1a9e627392c3b83e2fc236c13389005fa2cd
f619921d409a359d61c2103c9ae661db9910167436435a84b22773613724
550fe28950f2c83b8771f63afcd53ae"
}
```

可以看到，生成的地址是以 3 开头的多重签名地址。生成的地址以及 `redeemScript` 都需要保存好，以后通过多签地址来转账的时候会用到，其中 `redeemScript` 是赎回脚本，用来判断签名是否符合多签条件。

## 3. 从 3 开头的币地址向外发币

多重签名地址发起转账交易，与普通地址最大的不同就是需要多个签名来发起，这也是多签的一个目的所在。当使用一个钱包来通过多签地址向外发币的时候，如果多个地址的私钥都在钱包中，则使用过程与普通地址相同，如果私钥并不都在本地钱包中，则要生成交易后导出，并且让多方签名后才能发起交易。

接下来我们演示一下如何操作这种多签过程。为了方便，我们在比特币的测试网络中进行操作，其中的地址格式与主网中有所不同，操作过程是一样的。我们按照以下步骤来进行操作。



1) 打开 Bitcoin-qt 客户端, 并且连接 testnet 网络, 创建 3 个普通地址, 如下:

❑ msC9PzkGaH3aj5GVUi5BDCosYnfsWZvJXg

❑ mz5R1kdFdB9UM7nX49w2PSVlivfr2byHTb

❑ mm8RciNX62hHJPhic8iLhNq7JXjSnwnpaD

地址创建完毕后, 通过 faucet 网站获得一些测试用的比特币。进入到网址 <https://testnet.manu.backend.hamburg/faucet>, 分别给上述 3 个地址发送若干币。

2) 通过第 1 个和第 2 个地址创建一个多签地址, 签名条件为 2-2, 也就是必须 2 个私钥同时签名才能发起交易。我们使用 `addmultisigaddress` 命令来生成, 操作如下:

```
addmultisigaddress 2
['03998e2ef74a37361bcbf12d6d91d32ff1c4271bdaf19d43b145ebc46607c6f987',
'038420247121fbd14ccab8b9ead1b15d0d86d145c305556c9755f7f3ed3a746a59']'
```

执行命令后, 生成了一个多签地址, 如下所示:

```
2MxnDdYqRACKDF6dMSK2379Zulmoc45nULP
```

这里说明一下, `addmultisigaddress` 生成多签地址并且保存到了本地的钱包中, 同时我们看到生成的多签地址不是以 3 开头的, 因为我们在测试网络进行的操作。另外上述命令中的参数是公钥而不是地址, 这是可以通过 `validateaddress` 来查看的。创建多签地址后, 可以给多签地址也发送一些测试用的比特币。

3) 通过在本本地执行 `listunspent` 命令查看多签地址中的余额, 接下来就开始通过多签地址来发起一笔交易。我们看一下 `listunspent` 打印出来的多签地址的余额信息:

```
{
  "txid": "48c7846f486f2ee70d17ce163ac1817e8cb3e86f571c5e7ec8ff2ecdccd6ab71",
  "vout": 0,
  "address": "2MxnDdYqRACKDF6dMSK2379Zulmoc45nULP",
  "account": "",
  "redeemScript": "522103998e2ef74a37361bcbf12d6d91d32ff1c4271bdaf19d43b145ebc46607c6f98721038420247121fbd14ccab8b9ead1b15d0d86d145c305556c9755f7f3ed3a746a5952ae",
  "scriptPubKey": "a9143cb539ae820743aa90f244d60c5854f9786f889787",
  "amount": 0.65000000,
  "confirmations": 1,
  "spendable": true,
  "solvable": true,
  "safe": true
}
```

其中的 `txid` 信息可以用来构建多签地址的发起交易。发起交易如下:

```
createrawtransaction
['{"txid": "48c7846f486f2ee70d17ce163ac1817e8cb3e86f571c5e7ec8ff2ecdccd6ab71", "vout": 0}'],
{'mm8RciNX62hHJPhic8iLhNq7JXjSnwnpaD': 0.4,
'2MxnDdYqRACKDF6dMSK2379Zulmoc45nULP': 0.24999}'
```







```

    {
      "value": 0.24999000,
      "n": 1,
      "scriptPubKey": {
        "asm": "OP_HASH160 3cb539ae820743aa90f244d60c5854f9786f8897 OP_
          EQUAL",
        "hex": "a9143cb539ae820743aa90f244d60c5854f9786f889787",
        "reqSigs": 1,
        "type": "scripthash",
        "addresses": [
          "2MxnDdYqRACkDF6dMSK2379Zu1moc45nULP"
        ]
      }
    }
  ]
}

```

可以看到，输出了可读的 JSON 格式的解码数据。注意目前这个交易还是无效的，因为还没有完成签名，相当于是一个空白交易。从上述数据可以看到，字段 scriptSig 是空的。

4) 现在开始对交易进行签名。我们使用命令 signrawtransaction 来签名的。这个命令需要几个参数，我们先看一下这个命令的参数模式：

```

signrawtransaction "hexstring" ( [{"txid":"id","vout":n,"scriptPubKey":"hex","redeemScript":"hex"},...] ["privatekey1",...] sighashtype )

```

说明：

- 第 1 个参数是代签名交易的十六进制字符串；
- 第 2 个参数与创建交易参数意思相同，其中 scriptPubKey 是公钥脚本，redeemScript 是赎回脚本，这是多签地址发起交易必须提供的；
- 第 3 个参数是需要签名的私钥，我们的私钥都在钱包本地，因此可以不用显式提供；
- 第 4 个参数是签名类型。

参数中的 scriptPubKey 是需要获取的，通过如下命令可以获得：

```

getrawtransaction 48c7846f486f2ee70d17ce163ac1817e8cb3e86f571c5e7ec8ff2ecdccd6ab71 1

```

执行命令后，输出如下：

```

{
  "txid": "48c7846f486f2ee70d17ce163ac1817e8cb3e86f571c5e7ec8ff2ecdccd6ab71",
  "hash": "8e75b223c7a00e162e21b0ed9738bf02942dad33398f2bf0654429403755b18",
  "version": 1,
  "size": 247,
  "vsize": 166,
  "locktime": 0,
  "vin": [
    {

```





```

"txid": "af63e732eea4a78cb66782573eafda53fc0ba1b01bc232f62cf6fd025d1
960a4",
"vout": 1,
"scriptSig": {
  "asm": "00147df1b27d0cbd6bfdffbd8a617081580917643400",
  "hex": "1600147df1b27d0cbd6bfdffbd8a617081580917643400"
},
"txinwitness": [
  "304402204786823a0e19d0158951ed55e2dlac3d1e7fa6cc05c8e4b164ec10f
d167090a202204f7a81090b8ed3854022c0db71c94f21a4c791b03b0e41d8787
49d75fe4491a201",
  "030923e2572e19098b5eb6ab0b385902b69c9d27b37f19ed10ea59c473a3c8e9af"
],
"sequence": 4294967295
},
"vout": [
  {
    "value": 0.65000000,
    "n": 0,
    "scriptPubKey": {
      "asm": "OP_HASH160 3cb539ae820743aa90f244d60c5854f9786f8897 OP_
EQUAL",
      "hex": "a9143cb539ae820743aa90f244d60c5854f9786f889787",
      "reqSigs": 1,
      "type": "scripthash",
      "addresses": [
        "2MxnDdYqRACKDF6dMSK2379Zulmoc45nULP"
      ]
    }
  },
  {
    "value": 64.16679741,
    "n": 1,
    "scriptPubKey": {
      "asm": "OP_HASH160 327215db28da900463b0249df9d41ef25f6fe367 OP_
EQUAL",
      "hex": "a914327215db28da900463b0249df9d41ef25f6fe36787",
      "reqSigs": 1,
      "type": "scripthash",
      "addresses": [
        "2MwqxSfbkYasQcksfwRyWiMd5z5mg6ncghr"
      ]
    }
  }
],
"hex": "01000000000101a460195d02fdf62cf632c21bb0a10bfc53daaf3e578267b68ca7
a4ee32e763af01000000171600147df1b27d0cbd6bfdffbd8a617081580917643400ffffff
ff0240d2df030000000017a9143cb539ae820743aa90f244d60c5854f9786f8897873dc3767e
0100000017a914327215db28da900463b0249df9d41ef25f6fe367870247304402204786823a
0e19d0158951ed55e2dlac3d1e7fa6cc05c8e4b164ec10fd167090a202204f7a81090b8ed38

```



```

54022c0db71c94f21a4c791b03b0e41d878749d75fe4491a20121030923e2572e19098b5eb6
ab0b385902b69c9d27b37f19ed10ea59c473a3c8e9af00000000",
"blockhash": "000000032ffe12db6feec8315c728ed03634c2d7aad4fc476fdb2125775f5",
"confirmations": 10,
"time": 1521706049,
"blocktime": 1521706049
}

```

注意输出的信息，在 vout 段中，n 等于 0 的那一段，其中的 scriptPubKey 的 hex 属性就是需要放到参数中的值。至此，我们可以来执行 signrawtransaction 命令了，如下所示：

```

signrawtransaction "020000000171abd6cccd2effc87e5e1c576fe8b38c7e81c13a16ce170de72e6f486f84c7480000000000ffffffffff02005a6202000000001976a9143d8dfe9ebb3e93f64697897b6f09f5915cedd80e88ac58747d010000000017a9143cb539ae820743aa90f244d60c5854f9786f88978700000000"
'[{ "txid": "48c7846f486f2ee70d17ce163ac1817e8cb3e86f571c5e7ec8ff2ecdccd6ab71",
"vout": 0, "scriptPubKey": "a9143cb539ae820743aa90f244d60c5854f9786f889787" } ]'

```

执行成功后，输出如下：

```

{
  "hex": "020000000171abd6cccd2effc87e5e1c576fe8b38c7e81c13a16ce170de72e6f486f84c74800000000da00483045022100c941e73e2241f85a26ebe573033570bf6e83a61630c72b652228679478529cd102207b1f9a8e2c0453d83b0fb1f7a1780aa8c645a851c492a4704bdefecb2610779a0147304402201dddb36561fdaa877dcdf03070d87a30fb6bf1ce73352b3701ea3544598ea340220031668af875f2f258145ee92d163a3fcc81ecb180692a49bc70e9e6b772dae9b0147522103998e2ef74a37361bcbf12d6d91d32ff1c4271bdaf19d43b145ebc46607c6f98721038420247121fbd14ccab8b9ead1b15d0d86d145c305556c9755f7f3ed3a746a5952aefffffffff02005a6202000000001976a9143d8dfe9ebb3e93f64697897b6f09f5915cedd80e88ac58747d010000000017a9143cb539ae820743aa90f244d60c5854f9786f88978700000000",
  "complete": true
}

```

我们同样可以通过 decoderawtransaction 命令查看。可以发现 scriptSig 已经填上了值，说明已经有了签名信息。接下来还需要将交易广播到网络中，命令如下：

```

sendrawtransaction "020000000171abd6cccd2effc87e5e1c576fe8b38c7e81c13a16ce170de72e6f486f84c74800000000da00483045022100c941e73e2241f85a26ebe573033570bf6e83a61630c72b652228679478529cd102207b1f9a8e2c0453d83b0fb1f7a1780aa8c645a851c492a4704bdefecb2610779a0147304402201dddb36561fdaa877dcdf03070d87a30fb6bf1ce73352b3701ea3544598ea340220031668af875f2f258145ee92d163a3fcc81ecb180692a49bc70e9e6b772dae9b0147522103998e2ef74a37361bcbf12d6d91d32ff1c4271bdaf19d43b145ebc46607c6f98721038420247121fbd14ccab8b9ead1b15d0d86d145c305556c9755f7f3ed3a746a5952aefffffffff02005a6202000000001976a9143d8dfe9ebb3e93f64697897b6f09f5915cedd80e88ac58747d010000000017a9143cb539ae820743aa90f244d60c5854f9786f88978700000000"

```

如上所示，使用命令 sendrawtransaction 广播了交易数据。其中的参数就是之前的命令 signrawtransaction 执行成功后的 hex 输出值。广播执行完毕后，操作就完成了。有读者可





能会说，这个操作也太麻烦了吧，当然，这里只是用命令行来说明操作过程，实际可以使用对多签操作友好的一些钱包软件。

#### 7.4.6 隔离验证 SW 地址

隔离验证 (Segregated Witness) 是比特币的一种区块扩容方式，在比特币核心客户端的源码中，每个区块的大小限制为 1MB。当交易繁忙的时候，加上 1 个区块的生产间隔维持在 10 分钟左右，这样就会导致交易的网络拥堵。隔离验证是解决这个问题的一种方案。我们首先来了解一下隔离验证的概念。

在比特币的交易事务构造中，我们已经知道了，除了指定接收者和交易金额外，主要的数据就是签名验证信息。这些签名验证信息主要是用来验证转账者的比特币所有权。当然这个验证的工作主要是矿工节点要使用的，普通节点实际上并不关心与自己无关的交易中的签名信息。那么就出现了一种思路，将交易事务中包含的验证信息隔离出来，存放到另外一个地方，而在 UTXO 中只需要提供一个指向签名信息的指针即可，这样就能变相地对区块进行扩容了，类似于集装箱里面通过优化一下堆放的结构而省出空间的意思。由于这个修改改动到了交易签名数据，因此升级后，新版本与老版本的节点会有所差异，激活了隔离验证的节点可以生成 SW 地址。

可以用命令 `addwitnessaddress` 来生成一个隔离验证地址，需要一个现有的地址作为参数，当然前提是要在激活了隔离验证的客户端上才能成功执行。

### 7.5 比特币的生态系统

#### 1. 算力矿业

比特币是目前使用最为广泛的公有链系统，节点部署也相当广泛，由于具有特别的历史地位，因此其价值也是目前受到最广泛的认可。在比特币的整个生态系统中，算力矿业可以说是食物链的顶端，国内外都有很大型的比特币矿场。目前比特币的挖矿设备基本都是专业的集成电路矿机。由于比特币的价值越来越受到追捧，好的矿机也是“一机难求”，因此也造就了巨大的矿机市场。数量众多的矿机需要通过矿池服务来进行集群挖矿，因此也诞生了很多专业的矿池服务商。

从市场角度来看，比特币的挖矿主要在以下几个方面具备市场价值：

- 1) 矿机生产与销售；
- 2) 矿池服务；
- 3) 矿场建设与维护；
- 4) 多余电力的使用消耗。

不过这里也有一个潜在的问题。矿机一旦设计生产就只能针对比特币挖矿，对于共识算法与比特币不一样的其他区块链系统，矿机是不能通用的，因此一旦到了比特币新币奖

励越来越少的时候，整个基于比特币的算力矿业是不是能持续地发展下去，也是值得斟酌考虑的。

## 2. 交易平台

比特币是区块链技术的第一个应用，并且是作为一个电子现金系统的设计问世的。其带来的网络无边界操作特性和数据的不可篡改性以及匿名性等，都使得其在某种程度上得到越来越多的追捧，也因此产生了很多交易平台。国内外都有一些比特币交易平台。当然目前各个国家对于比特币或者说加密数字货币交易的政策也有差异，这毕竟涉及金融，国家必然是要站在宏观角度来管控的。中国从2017年9月4日后，关停了所有虚拟货币的交易平台。

## 3. 开发社区

相对比特币的交易，由比特币引领的开发社区是更有价值的。目前比特币的源码维护在GitHub站点之上，作为一个开源系统，对于今后的发展以及各种创新的和有趣的建议，都通过社区在维护。比特币有一个社区计划来维护它的升级和更新，叫BIP（Bitcoin Improvement Proposals），也就是比特币改进建议的意思。这是为比特币社区提供各种规范以及未来改进计划的设计指导文件，内容涉及比特币的方方面面，各种有意义的建议都可以提交上来，也因此而产生了很多新的开发内容，比如隔离见证方案、分层确定性钱包方案等，具体的内容大家可以到网址去查看：<https://github.com/bitcoin/bips>。除BIP外，目前还存在着不少的比特币开发社区，如比特币问答网站（StackExchange）、比特币对话发展和技术讨论论坛等，大家有兴趣的话可以多去参与这些开发社区，能了解到很多最新研究成果和动态。

除了对于比特币本身的发展社区外，还产生了基于比特币的一系列开发生态，比如各种类型的钱包、基于比特币的竞争币、基于比特币的存证系统等。

## 4. 币钱包和币应用

比特币的钱包有很多种类型，目前的种类如图7-16所示。



图 7-16 比特币钱包类型

图 7-16 是 bitcoin.org 站点上的截图，从中可以看到目前有各个平台上的钱包应用。实



际上除了这些平台上的，还有很多商业级的钱包应用。很多钱包功能被设计得更加智能化，集成了交易兑换的功能以及更丰富的交易转账操作。基于币的这种定义属性，也逐渐地出现了一些相关的应用，比如基于比特币的支付计算、商品交换市场等。当然在应用上的想象空间还是很大的。随着比特币的发展，未来有可能会更广泛的基于加密数字货币的衍生应用。

## 7.6 开发实施一个比特币存证应用

抛开比特币这种加密数字货币的特性，在技术上它就是一个分布式的、点对点的网络系统，并且在这个系统之上，通过工作量证明的共识机制以及特有的数据结构，实现了一些传统系统不具备的特性，其中就有不可篡改性以及不可伪造性，或者说这也是区块链技术的一个应用特性。并且，由于比特币是一个基于互联网的点对点的系统，没有第三方的权威干预，是一个相对公正透明的可信任环境。对于一个存证应用来说，什么是最重要的？我们可以看一下。

- 1) 数据的修改变动有时间戳的记录，并且不能抹除记录；
- 2) 每一次的操作行为都记录下来，形成数据历史；
- 3) 公正性，不存在一个第三方的强制干预；
- 4) 防止被恶意的篡改。

显然，区块链系统是能够实现这些要求的。作为节点部署最为广泛的比特币网络，是目前将区块链的技术特性发挥得最淋漓尽致的一个。接下来我们将实现一个简单的存证应用演示，在比特币网络存储一段信息的应用。

### 7.6.1 环境准备

开发环境选择 Ubuntu Desktop 操作系统，版本大于 14.04 即可。本机没有 Ubuntu 环境的，可以使用 VirtualBox 以虚拟机的形式来安装一个，这一部分的步骤就不再赘述了。系统环境准备好后，开始安装如下的组件。

#### 1. Python 开发环境安装

我们需要使用 Python2.x 环境，可以通过 Anaconda 来进行安装。这是一个 Python 的发行版，提供了包管理以及环境管理功能，使用很方便，而且可以同时管理多个 Python 环境，比如 Python2.x 与 Python3.x。可以到官网下载后安装，官网地址是 <https://www.anaconda.com/download/#linux>。

安装完毕后，可以创建一个 Python2.x 的专有环境。

```
conda create -n py2 python=2.7
```

通过以上命令就创建了一个 Python2.7 的环境，名称为 py2。我们可以在终端命令行中

通过如下命令来启动进入到这个 py2 的 Python 环境：

```
source activate py2
```

接下来，我们只需要安装一个 Python 的 bitcoin 调用库就可以了，命令如下：

```
pip install python-bitcoinlib
```

至此，基于 Python 的环境就安装好了。

## 2. 配置比特币测试网络

我们知道，比特币本身是有一个主网和一个测试网络的，不过为了便于开发测试，我们使用 Docker 镜像来安装一个比特币测试网络盒子 freewil/bitcoin-testnet-box。安装步骤如下：

```
sudo apt-get install docker.io
sudo service docker start
sudo docker pull freewil/bitcoin-testnet-box
```

第 1 行命令用来安装 Docker 环境，第 2 行命令是启动 Docker 服务，第 3 行命令是在 Docker 容器环境中安装比特币测试网络。安装完成后，可以试一下在 Docker 中启动 bitcoin-testnet-box。默认安装的 bitcoin-testnet-box 会启动 2 个节点，其中一个节点的 RPC 端口是 19001，另外一个节点的 RPC 端口是 19011。我们可以用使用如下命令启动 Docker 镜像：

```
sudo docker run -t -i -p 19001:19001 -p 19011:19011 freewil/bitcoin-testnet-box
```

如上命令，Docker 镜像已经启动成功，接下来我们就可以启动测试网络了。通过以下命令可以启动测试节点。启动成功后，在本机会模拟运行两个比特币测试节点，这两个节点就组成了一个最小的测试网络：

```
make start
```

我们可以看到有如如图 7-17 所示的执行输出：

```
bitcoind -datadir=1 -daemon
Bitcoin server starting
bitcoind -datadir=2 -daemon
Bitcoin server starting
```

图 7-17 输出结果

通过输出结果可以发现，这两个节点的数据目录分别在名称为 1 和 2 的目录中。

## 3. 测试网络使用

比起直接使用比特币的公共测试网络，在这样一个私有的测试网络中，很多操作是很方便的。我们可以来看一下两个网络节点的信息，执行命令如下：



```
make getinfo
```

可以看到如图 7-18 所示的输出结果：

```
bitcoin-cli -datadir=1 getinfo
[{"version": 130200,
  "protocolversion": 70015,
  "walletversion": 130000,
  "balance": 0.00000000,
  "blocks": 0,
  "timeoffset": 0,
  "connections": 1,
  "proxy": "",
  "difficulty": 4.656542373906925e-10,
  "testnet": false,
  "keypoololdest": 1521766306,
  "keypoolsize": 100,
  "paytxfee": 0.00000000,
  "relayfee": 0.00001000,
  "errors": ""}
]
bitcoin-cli -datadir=2 getinfo
{
  "version": 130200,
  "protocolversion": 70015,
  "walletversion": 130000,
  "balance": 0.00000000,
  "blocks": 0,
  "timeoffset": 0,
  "connections": 1,
  "proxy": "",
  "difficulty": 4.656542373906925e-10,
  "testnet": false,
  "keypoololdest": 1521766306,
  "keypoolsize": 100,
  "paytxfee": 0.00000000,
  "relayfee": 0.00001000,
  "errors": ""
}
```

图 7-18 网络节点信息

我们来介绍几个主要的属性信息：balance 是指钱包余额；blocks 是已经产生的区块数量，刚刚启动的时候都是 0；connections 是接入的其他节点的数量；paytxfee 是发送比特币的时候选择支付的手续费；relayfee 是每笔交易支付给矿工的默认手续费。

现在我们来试着产生一些区块看看。在比特币的主链和测试链中，大约是每 10 分钟产生一个区块，不方便进行测试开发。在现在这个模拟环境中，我们可以通过命令直接产生

区块。命令如下：

```
make generate
```

执行后可以立即模拟产生 1 个区块。如果要产生更多的区块，可以加一个参数。假如我们要直接产生 500 个区块，则可以如下操作：

```
make generate BLOCKS=500
```

注意，在以上命令产生区块的同时，就相当于确认了交易，并且钱包中会获得比特币奖励。在产生区块后，大家可以继续通过 `make getinfo` 命令查看节点状态变化。如果要新建钱包地址，可以通过 `make address1` 或者 `make address2` 来创建，分别是在第 1 个和第 2 个测试节点上创建。

## 7.6.2 示例程序

我们在这里实现一个简单的存证应用案例。将一段字符串存储到比特币账本，字符串是可以自行定义的。我们知道比特币中能够进行的操作主要就是转账交易，我们怎么在里面去存数据呢？实际上就是发起一笔交易事务。比特币支持一个指令码 `OP_RETURN`，可以在构造交易事务时，带上一小段自定义的文本数据而不是通常的交易数据，这样一来等于这笔交易事务就不能用来花费了，而只是作为一笔非交易的数据存储。不过，为了防止这样的非交易数据过多，所能携带的数据是受限制的，大约不能超过 40 字节。

现在我们知道了，实现一个存证应用，只要按照数据格式来构造一笔带有 `OP_RETURN` 标记的交易事务就可以了，整个过程与普通的交易处理是一致的，都包含从创建交易、签名到广播的过程。我们来看以下示例程序。

```
# -*- coding:UTF-8 -*-
```

```
from bitcoinrpc.authproxy import AuthServiceProxy, JSONRPCException
from decimal import Decimal
from binascii import hexlify, unhexlify
```

```
# 字节转十六进制
```

```
def byteToHex(value):
    hexValue=hex(value)[2:]
    return hexValue.zfill(2)
```

```
# 十六进制字符串按字节反转
```

```
def reverseHex(strHex):
    arr = []
    for s in range(0,len(strHex),2):
        arr.append(strHex[s]+strHex[s+1])
    arr.reverse()
```



```
# 设置 op return 输出脚本, 并替换掉普通的输出公钥脚本
```

```

newScriptPubKey = "6a" + hexlify(chr(len(savedata))) + hexlify(savedata)
tx = tx.replace(curScriptPubKey, newScriptPubKey)
# 编码为原始交易事务格式
rpc.decoderawtransaction(tx)

# 签名并且广播交易事务
tx = rpc.signrawtransaction(tx)['hex']
rpc.sendrawtransaction(tx)

# 打印交易事务
print tx

```

通过示例代码我们可以看到，主要过程就是创建带有 `op_return` 标记的交易事务。创建方式有很多种，这里使用了 Python 通过 `python-bitcoinlib` 来构建。感兴趣的读者也可以直接通过字符串拼接的方式来构建原始交易事务。

## 7.7 小结

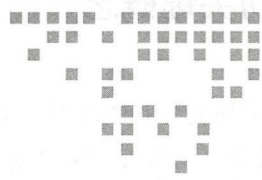
作为区块链技术的首个系统实现，也作为第一个将区块链技术带入到世人眼前的系统，比特币拥有着相当重要的历史地位，也有着很典型的研究价值。后续出现的各种其他区块链系统，都是以比特币的技术栈为基础展开的，因此完整而深入的理解比特币的设计，对于理解区块链系统的构造原理大有裨益。其中除了技术实现的部分，包含的经济模型设计也是很值得去研究的，挖矿奖励、交易手续费、产量缩减等，都是很有意思的经济模型设计。

通过对比比特币技术的了解，我们可以知道，一个典型的区块链系统基本上是由区块数据存储、共识机制、密码算法、脚本引擎以及 P2P 网络处理模块这几个部分组成的。比特币通过组合这些技术实现了一个点对点的电子现金系统，其中各种具体的技术细节都是在这几个部分的基础上展开的。大家在学习比特币的时候，一定不要只是着眼于某个细节技术的实现，一定要放到一个整体中来看待：一个分布式的基于 P2P 网络的电子现金系统，是如何通过区块链技术来构造实现并且能解决诸如双花、网络自治的问题的。

### 习题

- [1] 比特币钱包必须要携带完整账本数据才能工作吗？
- [2] 比特币的发行总量是确定的 2100 万个吗？
- [3] 区块之间是通过什么来链接的？
- [4] 挖矿算法中使用了怎样的哈希计算？
- [5] 比特币地址是通过什么生成的？
- [6] 外部程序如何访问比特币节点？





## 第 8 章 Chapter 8

## 以太坊

作者：庄鹏

比特币网络自 2009 年出现以后，这种利用脚本编程的加密货币，基于工作量证明打包交易并铸币发行的公有区块链网络，虽然历经几次大动荡和调整，但因其架构设计的合理性，获得全球社区的支持，成为加密货币的领头羊。但是比特币只能基于脚本和少数的指令进行业务编程，无法实现复杂的业务逻辑。以太坊的创始人 Vitalik Buterin、Gavin Wood 和 Jeffrey Wilcke 在对加密货币进行长期研究后，在已有的可编程的加密货币和区块链技术基础上增加了一个虚拟机。这个虚拟机支持更加复杂的指令运行，并且达到了图灵完备，这样任何应用需求都可以被转化为可以由计算机执行的指令。为了限制对虚拟机和存储的使用，开发者还设计了虚拟机上执行应用的经济成本（Gas 消耗）。另外由于提高了产块速度，挖矿更容易产生废块，以太坊对叔区块也引入了奖励，以此增强了网络算力的安全性。相对于比特币，以太坊基于以太坊基金会这种更加中心化的开发组织领导和维护着整个网络的运行、升级；开发人员基于高级的编程语言，可以编写图灵完备的应用程序。

高级语言编写的程序经过编译后，二进制代码可以在以太坊虚拟机上运行。可以说，以太坊把只能进行少量脚本编程的加密货币比特币，拓展成了可以运行任意应用的世界计算机。由于应用的运行结果可以为联网的世界上任意参与方的验证节点所验证和共同认可，那么按照预先设定的规则运行的应用程序就变成了一种机器契约程序，只要预先设定的规则和条款满足就会执行预先设定的动作，这样的机器契约程序就叫智能合约（Smart Contract）。这个概念早在 1995 年就由 Nick Szabo 提出<sup>①</sup>，可以说以太坊第一次实现了这种概念。

---

① Nick Szabo. Formalizing and securing relationships on public networks. First Monday, 2(9), 1997.

## 8.1 以太坊关键概念

### (1) 账户

作为内置了加密货币的系统，以太坊同比特币的单一 UTXO 模型不一样，具有账户 (Account) 的概念。以太坊的账户分为外部账户<sup>①</sup> (Externally Owned Accounts, EOA) 和合约账户 (Contract Accounts)。账户为密码学意义上的账户，跟比特币一样控制了私钥就控制了账户的操作权。

1) 外部账户为一般意义上的用户账户，用户在创建账户时自动生成公私钥对，编码存放在 Keyfile 中，私钥使用用户口令加密，公钥哈希值截取后 20 位作为账户地址。

2) 合约账户保存在以太坊区块链上，是合约代码 (功能) 和数据 (状态) 的集合。合约账户通过外部账户或既存合约账户进行部署和操作控制。在部署合约二进制代码时由以太坊虚拟机 (EVM) 基于创建者账户地址、创建者交易 nonce 生成账户地址。账户地址生成过程如图 8-1 所示。

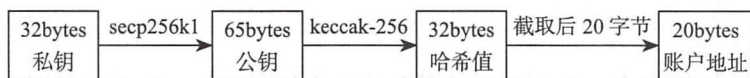


图 8-1 账户地址生成过程

### (2) 交易

以太坊的交易 (Transaction) 是外部账户发往其他账户 (外部账户或合约账户) 报文结构体，主要包括接受者地址 (0 地址代表新建合约实例)、发送者签名、发送金额、数据域 (若接收方为合约账户)、Gas 上限和 Gas 价格。

### (3) 消息

以太坊的消息 (Message) 是合约账户发往其他账户 (外部账户或合约账户) 报文结构体，主要包括消息发送者、消息接受者地址、发送金额、数据域 (若接收方为合约账户) 和 Gas 上限。

### (4) Gas

由于以太坊为公有链设计，为了避免恶意程序无代价发动 DoS 攻击或滥用以以太坊网络，参考比特币引入了经济概念，所有在以太坊 EVM 上的操作和存储消耗都需要 Gas。交易发送者设定 Gas 上限和 Gas 价格作为矿工打包费用。如果费用过低，矿工可以选择不打包交易到区块。

合约在执行过程中，如果当前代码执行以及子消息产生的代码执行发生的所有 Gas 消耗超过 Gas 上限，矿工依然可以获得打包费用，但是当前交易的所有状态都会被回滚。如果未出现 Gas 耗尽 (gas overrun)，交易成功执行，所有未消耗的 Ether (以太币) 依然会返

<sup>①</sup> Ethereum Homestead Documents. Contracts and Transactions. 2016. URL {<http://www.ethdocs.org/en/latest/contracts-and-transactions/account-types-gas-and-transactions.html>}.



回发送者账户。

矿工挖矿获得 Ether，合约执行消耗 Ether，形成天然的供求关系。Gas 只与程序逻辑处理复杂度有关，将其与市场波动相关的 Ether 价格隔离开，通过发送者设定 Gas Price（燃料价格），以太坊设计了一套对抗通货紧缩 / 膨胀的方法。

### （5）合约

以太坊的合约（Contract）即为合约账户，是以 EVM 字节码形式存储的代码（功能）和数据（状态）的集合体。它们以 Patricia 树组织，存储在区块链的账户地址上，账户可以相互发送交易（Transaction）或消息事件（Message Event），在 Gas 上限内进行一次交易的图灵完备运算。交易发送者支付“实际消耗 Gas \* 指定 GasPrice”的 Ether 给矿工作为费用，将此交易和结果打包到下一个区块上。图 8-2 为智能合约间的交互示意图。

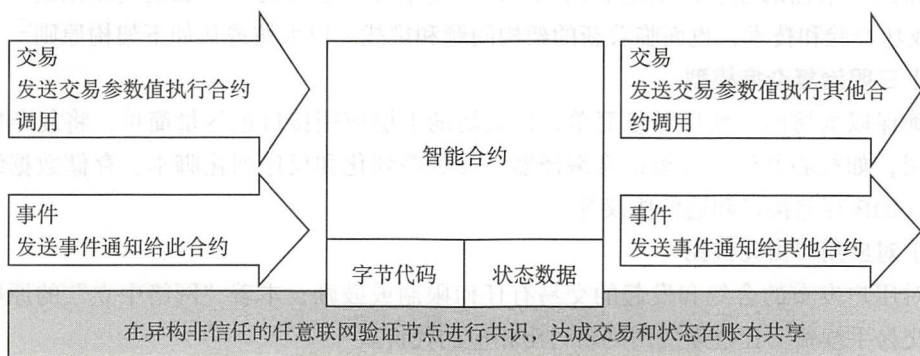


图 8-2 智能合约间的交互

### （6）以太坊虚拟机

以太坊虚拟机（EVM）是智能合约的运行环境，是比 JVM 或 Docker VM “沙箱”模式还严格的一种完全隔离的虚拟机运行环境。运行在 EVM 中的合约代码无法访问网络、文件系统和其他进程，合约之间的访问也受到严格限制。EVM 是基于栈操作的虚拟机，栈最大深度 1024，每个元素 32 字节，其指令集由最基本的算术、位、逻辑、比较、条件跳转、无条件跳转这些满足图灵完备的最小指令集组成，EVM 可以发送消息调用其他合约，还可以加载目标地址的合约代码在当前合约上下文执行（callcode）。

### （7）DApp

DApp 直接关联起用户和服务提供方，即买卖双方的应用。以太坊提供多种语言版本的 API 客户端，并通过以太坊客户端接入网络，DApp 基于这些 API 客户端，把需要共识的业务逻辑和数据以合约的形式部署到以太坊区块链上，并按照需要执行 API 调用操作。那些不需要共识的逻辑在 DApp 自身中实现，以降低运行成本。表 8-1 为以太坊 DApp API 库。

表 8-1 DApp API 库

| API 库   | 语言         | 项目地址  |
|---------|------------|---|
| web3.js | JavaScript | <a href="https://github.com/ethereum/web3.js">https://github.com/ethereum/web3.js</a> |

(续)

| API 库         | 语言      | 项目地址  |
|---------------|---------|---|
| web3j         | Java    | <a href="https://github.com/web3j/web3j">https://github.com/web3j/web3j</a>                             |
| Nethereum     | C# .NET | <a href="https://github.com/Nethereum/Nethereum">https://github.com/Nethereum/Nethereum</a>             |
| ethereum-ruby | Ruby    | <a href="https://github.com/DigixGlobal/ethereum-ruby">https://github.com/DigixGlobal/ethereum-ruby</a> |

## 8.2 以太坊的架构

作为分布式价值交换网络，以太坊要建模的是一个公有的、可以被世界上任何一个联网的人使用的状态机。这个状态机由交易驱动，在其上可以执行图灵完备的应用，可以自动在分布式不信任的互联网上达成共识。以太坊借鉴了已有的一些加密货币系统（如比特币）的成功经验和技术，也面临着新的架构问题和挑战。以太坊遵从如下架构原则<sup>①</sup>。

### （1）三明治复杂度模型

为确保以太坊网络底层尽量简单，以太坊的上层应用接口也尽量简单，将复杂度集中在中间层，如核心共识、高级语言编译器、参数序列化和反序列化脚本、存储数据结构模型、LevelDB 存储接口和通信协议等。

### （2）对应用自由无偏见

不对用户发布的合约和发起的交易有任何限制或鼓励，本着“网络中立”的原则，以太坊“交易手续费”让过度使用网络付出相应的代价。

### （3）协议和操作码通用性

所有协议特性和 EVM 操作码都包含尽量少的语义含义，这样可以充分组合使用，满足未来更高级的语义用途。如将“消息”（message）拆分成“功能调用”（function call）和“外部观察者感兴趣的事件”（event interesting to outside watchers）两个概念。例如使用 LOG 操作码记录仅仅“喂给”（feed）DApp 感兴趣的信息，而不是把所有的交易和消息都记录下来。

### （4）没有提供专门的特性

通用性设计的必然结果就是平台的协议不提供任何特性，哪怕是非常高层次的用例。用户可以把这些特性通过智能合约的方式实现，如特定的合约代币（英文为 token，又译为通证<sup>②</sup>）、侧链。反例是比特币提供的锁定时间（locktime）特性，在以太坊中完全可以基于合约实现这种特性。

### （5）有风险偏好

对于存在引起风险的技术改变，如果可以带来很大的收益，如极大地提高出块速度、提高共识效率，都是可以接受的。

<sup>①</sup> 参阅 <https://github.com/ethereum/wiki/wiki/Design-Rationale>。

<sup>②</sup> 孟岩，《通证是下一代互联网数字经济的关键》，2017。



## 8.2.1 以太坊数据模型

### 1. RLP 编码

RLP 编码全称是 Recursive Length Prefix (递归长度前缀), 该算法可以用来序列化任意嵌套的二进制数组, 是以太坊主要的序列化编码算法<sup>①</sup>。

编码对象基本元素 (item) 可以是一个字符串, 也可以是由若干 item 组成的列表。RLP 按照如下编码规则和伪代码进行算法编码。

编码规则:

- 1) 单个 ASCII 字节  $\in [0x00, 0x7f]$ , 编码是其本身。第一个字节  $\in [0x00, 0x7f]$ ;
- 2) 字符串长度  $\leq 55$ , 编码是  $(0x80 + \text{长度}) + \text{字符串}$ 。第一个字节  $\in [0x80, 0xb7]$ ;
- 3) 字符串长度  $> 55$ , 编码是  $(0xb7 + \text{长度的二进制的字节数}(1 \sim 8)) + \text{长度} + \text{字符串}$ 。第一个字节  $\in [0xb8, 0xbf]$ 。

若对象是列表, 对列表项依次进行 RLP 编码, 顺序拼接成列表的 payload:

- payload 长度  $\leq 55$ , 编码是  $(0xc0 + \text{payload 长度}) + \text{所有列表项的 RLP 编码}$ 。第一个字节  $\in [0xc0, 0xf7]$ ;
- payload 长度  $> 55$ , 编码是  $(0xf7 + \text{payload 长度的二进制的字节数}(1 \sim 8)) + \text{payload 长度} + \text{所有数字项的 RLP 编码}$ 。第一个字节  $\in [0xf8, 0xff]$ 。

RLP 的伪代码如图 8-3 所示。

```
def rlp_encode(input):
    if isinstance(input, str):
        if len(input) == 1 and ord(input) < 0x80: return input
        else: return encode_length(len(input), 0x80) + input
    elif isinstance(input, list):
        output = ''
        for item in input: output += rlp_encode(item)
        return encode_length(len(output), 0xc0) + output

def encode_length(L, offset):
    if L < 56:
        return chr(L + offset)
    elif L < 256**8:
        BL = to_binary(L)
        return chr(len(BL) + offset + 55) + BL
    else:
        raise Exception("input too long")

def to_binary(x):
    if x == 0:
        return ''
    else:
        return to_binary(int(x / 256)) + chr(x % 256)
```

图 8-3 RLP 伪代码

① 推荐阅读 <https://github.com/ethereum/wiki/wiki/RLP>。

编码格式举例如下：

```
"dog" = [ 0x83, 'd', 'o', 'g' ]      // 字符串
// 字符串列表
[ "cat", "dog" ] = [ 0xc8, 0x83, 'c', 'a', 't', 0x83, 'd', 'o', 'g' ]
('null') = [ 0x80 ]      // 空串
[] = [ 0xc0 ]      // 空数组
0 = [ 0x80 ]      // 整数 0
('\x00') = [ 0x00 ]      // 整数 0 Hex 编码
('\x0f') = [ 0x0f ]      // 整数 15 Hex 编码
('\x04\x00') = [ 0x82, 0x04, 0x00 ] // 整数 1024 hex 编码
[ [], [[]], [ [] ], [[]] ] = [ 0xc7, 0xc0, 0xc1, 0xc0, 0xc3, 0xc0, 0xc1, 0xc0 ]
// 嵌套空列表
"Lorem ipsum dolor sit amet, consectetur adipiscing elit" = [ 0xb8, 0x38, 'L',
'o', 'r', 'e', 'm', ' ', ... , 'e', 'l', 'i', 't' ]
```

## 2. Merkle Patricia 树

改进版 Merkle Patricia 树<sup>①</sup> (Merkle Patricia Tree/Trie) 是以太坊主要的数据结构，用来存储 (key, value) 键值对，可用于账户状态、交易列表、收据列表等类似字典 (dictionary) 的数据结构，并进行树形结构的数据表示和存储，对于插入、检索、修改、删除可以实现  $O(\log(n))$  搜索复杂度。Merkle Patricia 树是由 Alan Reiner 提出并在 Ripple 协议上实现，是基于 Merkle 树 ([https://en.wikipedia.org/wiki/Merkle\\_tree](https://en.wikipedia.org/wiki/Merkle_tree)) 和 Radix 树 ([https://en.wikipedia.org/wiki/Radix\\_tree](https://en.wikipedia.org/wiki/Radix_tree)) 的一种组合数据结构，可以降低 Radix 树存储空间需求，提高树搜索时效性，并可以保证基于相同的 key/value 键值对的集合构建出的 Patricia 树的各节点每个字节都相同，即根哈希值也相同，任何对键值对的调整都会导致不同的根哈希值计算结果。

以太坊的 Patricia 树用来唯一计算各种字典型数据，如合约状态存储根哈希、世界状态根哈希，以及计算各种列表型数据（如区块交易列表，交易收据列表）。以太坊 keccak-256 哈希算法计算出来交易的哈希值都是 256 bit，即 32 个字节，如果用十六进制表示，就是 64 个字符，每个字符代表 4bit，称为一个 nibble，一般都是将哈希值的 64 个字符作为索引 key，value 可以是任意长度的二进制。为了说明 Merkle Patricia 树的结构，以一个简化的余额状态集合的构建进行说明，实际的状态 Patricia 树是 key 为 sha3 (ethereumAddress)，value 为 RLP ([nonce, balance, storageRoot, codeHash]) 的所有账户 (EOA 和合约账户) 的集合。假设要表示一组 (address, balance) 键值对的状态集合：[(a711355, 45.0ETH), (a77d337, 1.00WEI), (a7f9365, 1.1ETH), (a77d397, 0.12ETH)]，基于 Merkle Patricia 树会构建出如图 8-4 所示的数据结构<sup>②</sup>。

① 推荐阅读 <https://github.com/ethereum/wiki/wiki/Patricia-Tree>。

② 参见 <https://ethereum.stackexchange.com/questions/268/ethereum-block-architecture>。



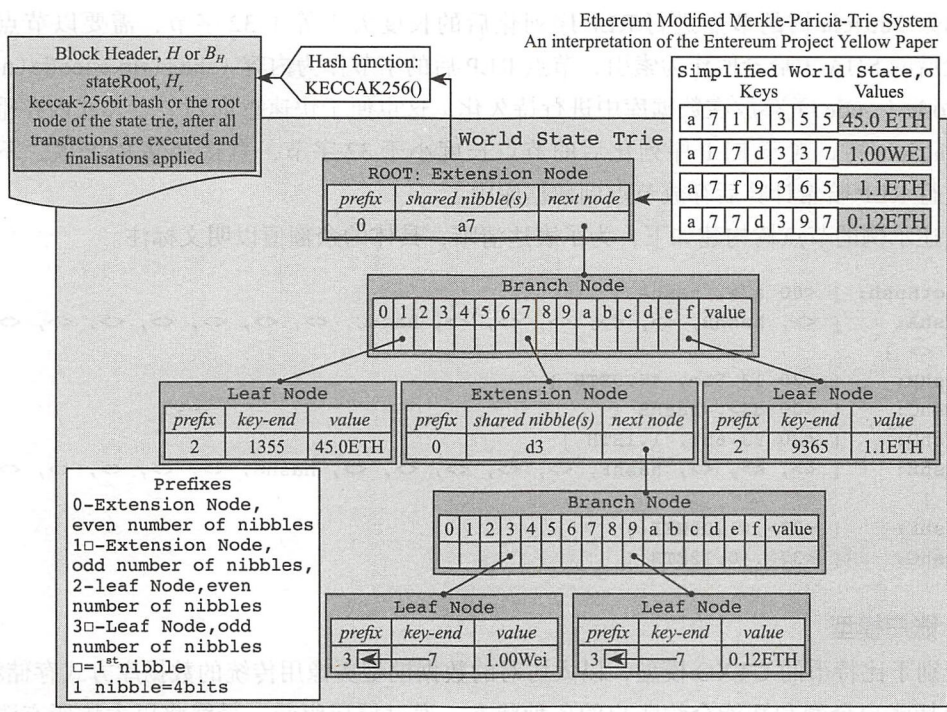


图 8-4 Merkle Patricia 树构建过程示例

Merkle Patricia 树在 Radix 树的基础上引入了一些复杂的数据结构，以降低存储消耗并提高检索的效率。一个 Patricia 树包括以下几种节点。

- 1) 分支节点 (Branch): 一个由 17 个元素组成的节点  $[v_0 \dots v_{15}, v_i]$ 。
- 2) 叶子节点 (Leaf): 一个由 2 个元素组成的节点  $[\text{encodedPath}, \text{value}]$ 。
- 3) 扩展节点 (Extension): 一个由 2 个元素组成的节点  $[\text{encodedPath}, \text{hash}]$ 。

扩展节点 encodedPath 代表至少 2 个 nibble，并且至少有两个要存储的 key，经过这个扩展节点进行遍历。

由于一个字节 (8bit) 可以存储 2 个 4bit 的 nibble，这样就导致无法区分如 nibble “1” 和 nibble “01”，因为都存成 0x01，为了区分 nibble 奇偶数的情形，以及区分都由 2 个元素组成的叶子节点和扩展节点，特别设计了一种前缀编码和压缩算法，如表 8-2 所示。

表 8-2 Merkle Patricia 树节点前缀编码

| 节点区分编码 (第 1 字节高位) | 节点类别      | nibble 长度奇偶性 | 第 1 字节低位               |
|-------------------|-----------|--------------|------------------------|
| 0                 | 扩展节点      | 偶数           | 补 0                    |
| 1                 | 扩展节点      | 奇数           | 1 <sup>st</sup> nibble |
| 2                 | 叶子节点 (结束) | 偶数           | 补 0                    |
| 3                 | 叶子节点 (结束) | 奇数           | 1 <sup>st</sup> nibble |

如果 Patricia 树的节点进行 RLP 序列化后的长度大于等于 32 字节，需要以节点 RLP 序列化后的 SHA-3 哈希值作为索引，节点 RLP 后的字节作为内容（`sha3(rlp.encode(node))`，`rlp.encode(node)`）存放在数据库中进行持久化，这也便于快速查询到节点并在内存重新构建出 Patricia 树，对于 RLP 序列化后的节点长度小于 32 字节，直接存储 RLP 值，不需要另外进行 RLP 哈希计算后存放节点到数据库中。

上述示例的节点树构建如下，为了表达清晰，具体的余额值以明文标注。

```

rootHash: [ <00 a7>, hashA ]
hashA:    [ <>, hashB, <>, <>, <>, <>, <>, hashC, <>, <>, <>, <>, <>, <>, <>, <> ]
hashD, <> ]
hashB:    [ <20 13 55>, 45.0ETH ]
hashC:    [ <00 d3>, hashE ]
hashD:    [ <20 93 65>, 1.1ETH ]
hashE:    [ <>, <>, <>, hashF, <>, <>, <>, <>, <>, hashG, <>, <>, <>, <>, <>, <> ]
hashF:    [ <37>, 1.00WEI ]
hashG:    [ <37>, 0.12ETH ]

```

### 3. 账户模型

区别于比特币的 UTXO 模型，以太坊对的数据模型更像用传统的数据库方式存储状态，状态包括账户余额和智能合约账户的各种状态，通过层层组装，最终将以太坊所有账户和对应的状态（包括合约存储子状态哈希）用 Patricia 树的方式组织成账户模型。

为什么采用账户模型而不是比特币的 UTXO 模型，这是由以太坊建模状态的复杂度决定的。由于在以太坊虚拟机上运行的程序非常复杂，每一步指令的运行都需要消耗 Gas 进而消耗交易发起者的加密货币，为了提高网络攻击成本，运行不成功的交易所消费掉的 Gas 是不被返还的。如果采用 UTXO 模型，以太坊支持“世界状态”，需要构建状态根（state root）类似的 UTXO，对一个包含若干指令的交易的运算验证将非常复杂，且交易会影响到若干账户的余额和合约账户的若干状态，这不仅涉及金额状态的 UTXO 验证，还需要处理合约账户状态数据的 UTXO，区块包括的所有交易都需要依次执行这种基于状态根类似 UTXO 的验证，验证复杂度会非常高，所以以太坊采用了较为简单，也更集中化的传统账户模型来记录每个地址账户的余额。

另外，采用账户模型，对于同一个账户地址相关的多个交易，还可以减少 UTXO 方式重复存储 20 字节地址和 32 字节交易 ID 在所需的空間，账户模型只需要存储 20 字节地址和 8 字节余额以及 2 字节的 nonce 即可。

账户模型也有利于降低轻客户端合约账户内数据状态查询和验证复杂度，轻客户端只需访问状态 Patricia 树即可，不需要长时间操作状态根类似的 UTXO 进行查询和验证。

以太坊设计为传统的账户数据模型，所有账户（EOA 和合约账户）都需要保存余额（balance，以太币 Wei 为单位）。另外，对于合约账户，还需要保存合约账户的各种状态值，把合约账户中所有要记录的状态和对应的值以 Patricia 树方式构建得到合约账户的状



态存储根哈希值 (storageRoot)，对于外部用户账户 EOA，这个值就是空字符串。以太坊会将所有账户 (EOA 和合约账户) 的余额 (balance)，账户交易序列号 (nonce)，账户的状态存储根哈希值 (storageRoot)，合约代码哈希值 (codeHash) 组成的元组 [nonce, balance, storageRoot, codeHash] 进行 RLP 序列化编码的字符串作为 value，以对应账户的 20 个字节账户地址作为 key，再以 Patricia 树进行组织，计算出世界状态根哈希值，作为当前区块头哈希值的计算要素之一，详细的区块头哈希值的计算数据结构见下一节说明。

#### 4. 区块结构

以太坊一个区块的构成结构如下<sup>①</sup>：

```
[
  blockHeader,
  uncleHeaderList,
  transactionList,
]
```

一个区块由区块体 [uncleHeaderList, transactionList] 和区块头 blockHeader 构成。

区块体各部分的内容组成如下。

uncle HeaderList 是叔区块头列表。

```
uncleHeaderList = [
  uncle_block_header_1,
  uncle_block_header_2
]
```

由于以太坊出块设计成平均 14 秒一个区块，因网络传播不及时导致废块率大大提高，从而网络的安全性降低，为了提高网络安全性，Yonatan Sompolsky 和 Aviv Zohar 在 2013 年 12 月首次提出了 Greedy Heaviest Observed Subtree (GHOST) 协议，把最多往前 6 个区块的“叔区块”纳入当前区块中，每纳入一个叔区块就可以额外得到 1/32 个固定 Ether 的收入，每个区块最多包括 2 个叔区块。基于 PoW 的矿工挖矿，除获得 5Ether 的固定挖矿收入之外，还可以额外获得打包叔区块的收益，经济上也有动力。而整个网络由于包括了叔区块，被打包的叔区块矿工会得到 7/8 个固定 Ether 的收入，整个网络的哈希算力更大，网络会更安全，另外废块矿工也将获得 Ether 收入，这样拥有大哈希算力的矿池不至于越来越垄断，产生算力中心化的风险。

交易列表是当前打包进区块的所有交易。一个交易可以是外部账户的转账交易，也可以是合约部署交易，还可以是合约函数的执行交易。下面是一个交易列表示例：

transactionList 是当前区块的交易列表：

```
transactionList = [
  transaction 0,
```

① 推荐阅读 <https://github.com/ethereum/wiki/blob/master/Block-Protocol-2.0.md>。

```
transaction 1,
...
]
```

区块头哈希值的计算较为复杂，具体结构如下所示：

```
blockHeader = [
    Parent Block Hash, # 上一个区块头哈希
    Uncle Header List Hash, # 叔区块头列表 trie hash
    Coinbase Address, # 矿工挖矿收益账户地址
    State Root Hash, # 合约 Patricia 树根哈希
    Tx List Hash, # 挖矿交易列表 trie hash, rlp(transactionIndex) 为 key
    Receipt List Hash, # 每个挖矿交易收据列表 trie hash
    Bloom, # Bloom 过滤器用来验证日志已经存在①
    Difficulty, # 当前难度值
    Block Number, # 当前区块高度
    GasLimit, # 当前区块 Gas 消耗限制
    GasUsed, # 当前区块使用 Gas
    Time, # 挖矿时间
    Extra, # 挖矿设定额外数据
    MixDigest, # 基于 DAG 计算出的哈希值，同 nonce 一起作为挖矿尝试因子
    Nonce # 同 MixDigest 一起作为挖矿尝试因子
]
```

区块头哈希值的组成部分都有说明，不难理解。这里仅介绍其中的一个数据结构 Bloom。

Bloom 过滤器是由 Burton Howard Bloom 在 1970 年提出的一种节省空间和时间的概率数据结构，用以检查一个元素是否在一个大的集合中。以太坊使用 Bloom 过滤器来检测合约事件的发生。合约在执行交易被打包进区块时，合约地址及发送事件的最多 3 个用户索引主题和事件方法名的 hash 主题会一起加入到 Bloom 过滤器中。如果轻客户端启动对合约的某个事件的监听，通过查找区块的 Bloom 过滤器即可知道当前监听的事件是否发生。区块头结构和生成过程如图 8-5 所示。

图 8-5 由下而上各层次的计算过程如下。

1) 合约的状态存储数据也以 Patricia 树进行组织，构建出的 Patricia 树的根进行 Keccak-256 哈希得到 storage root hash。

2) 所有账户的 [balance, nonce, storage root hash, code hash] 进行 RLP 编码序列化后，同 20 字节账户地址组成键值对，一起构建状态 Patricia 树，对状态 Patricia 树的根进行 Keccak-256 哈希计算后，得到状态根哈希值。整个状态 Patricia 树存放在本地，只把状态根哈希值放在区块头上，参与计算整个区块的哈希值。

3) 状态根哈希值同父区块哈希值、叔区块头列表的 trie hash、矿工地址、打包进区块的交易列表 trie hash、收据列表 trie hash、Bloom 过滤器（2048bit）、难度值、当前区块号、

① 推荐阅读 [https://en.wikipedia.org/wiki/Bloom\\_filter](https://en.wikipedia.org/wiki/Bloom_filter)。



区块消耗 Gas 上限、区块所有交易消耗 Gas、时间戳、区块额外数据、挖矿因子 MixDigest 和 nonce 一起进行 RLP 序列化编码后, 执行 Keccak-256 哈希函数得到这个区块的哈希值。其中, 所有列表型的数据结构体, 如叔区块头列表、交易列表、收据列表, 都以 Patricia 树的形式进行构建, 构建以列表序号作为 key, 列表的内容作为值进行构建。

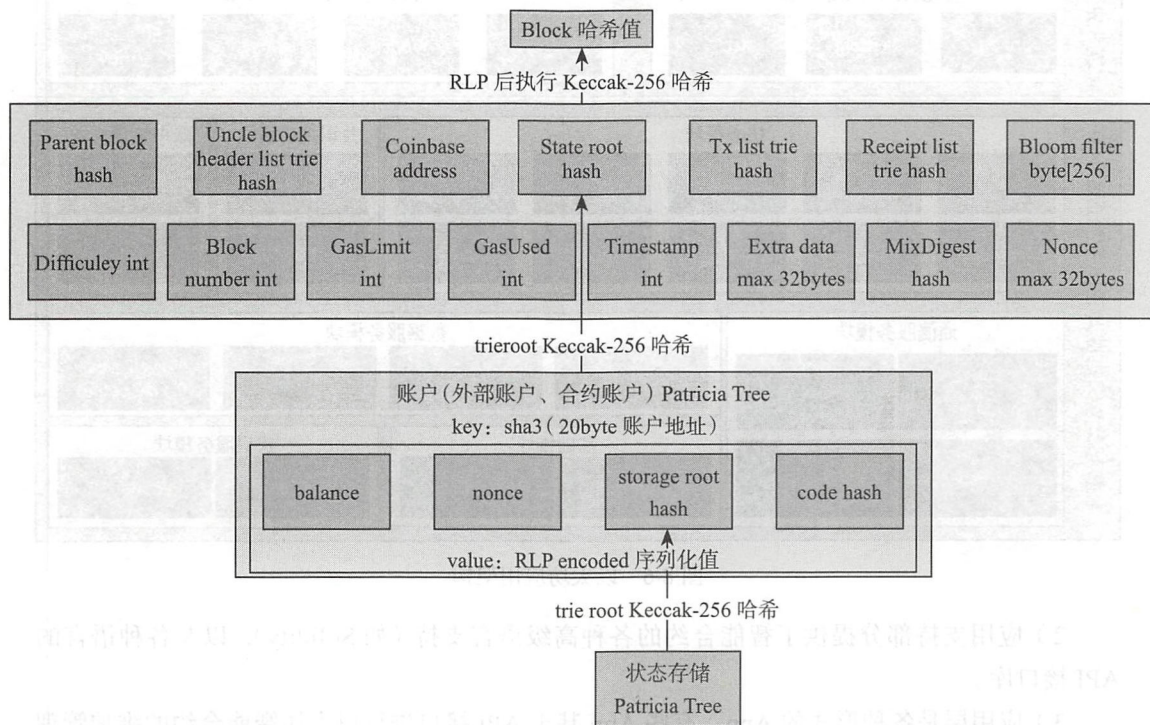


图 8-5 区块头结构和生成过程

## 8.2.2 以太坊的应用架构

基于前面的架构原则和选择的数据模型以及各种算法, 以太坊基金会建设了一个较为完备的生态系统, 其中包括多种语言实现的核心网络、多种智能合约编写的高级语言支持编译器、多种语言的 SDK 和 API 接口、Mist 钱包浏览器、ENS 以太坊域名服务注册 DApp。其他公司和个人也开发了很多合约部署在以太坊网络上运行。图 8-6 所示为以太坊应用架构图。

以太坊平台分成大致 3 个层次。

1) 基于黄皮书<sup>①</sup>的以太坊客户端或部分组件的各种语言实现 (Go, C++, Python, Java, Ruby, Rust, JavaScript), 这些客户端在世界范围内构成了以太坊的区块链网络。

① Gavin Wood. Ethereum yellow paper. 2015. URL {<http://gavwood.com/Paper.pdf>}.

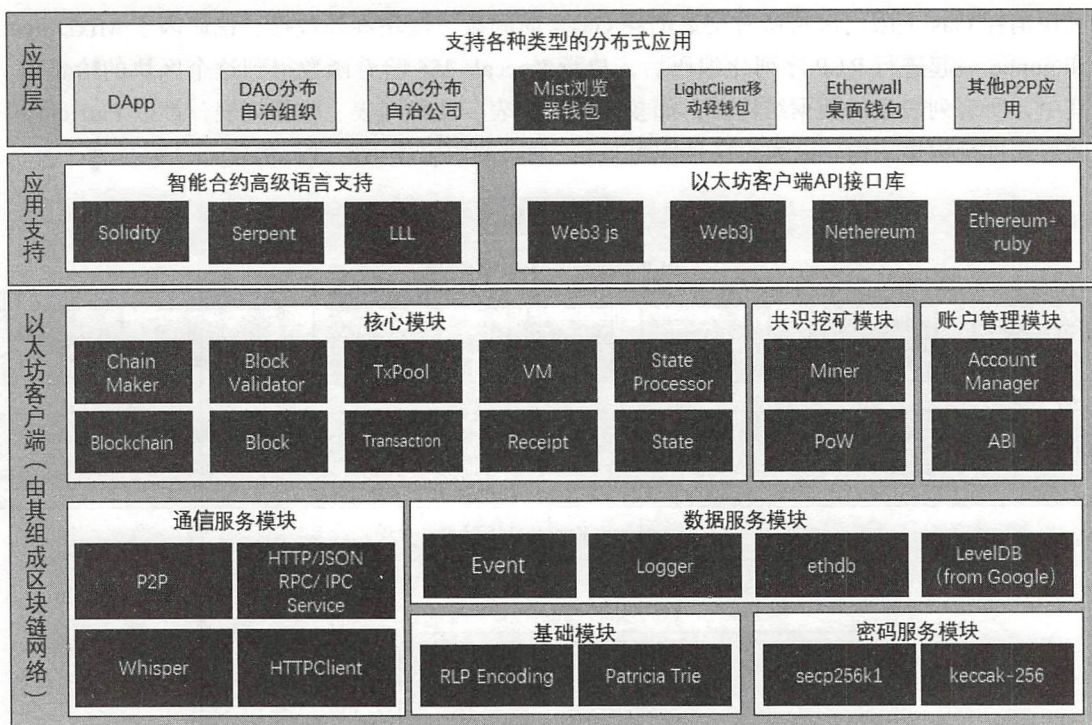


图 8-6 以太坊应用架构

2) 应用支持部分提供了智能合约的各种高级语言支持（如 Solidity），以及各种语言的 API 接口库。

3) 应用层是各种形式的 App：有些 App 基于 API 接口进行以太坊智能合约的维护管理操作，有些 App 基于智能合约提供复杂的应用功能，有些 App 作为钱包可以执行 Ether 的余额查看和转账，等等。

以太坊作为一个分布式共识状态机，其核心功能组成包括：状态表示、驱动交易、转换引擎、通信服务和共识验证。

#### （1）状态表示

每个验证节点在执行相同时序的任意一笔交易后的结果，都有相同的状态表示。以太坊要在所有节点形成的共识是账户信息，这包括所有账户的余额、每个账户当前执行 nonce，如果是合约账户，还要记录合约的当前存储状态、合约的二进制执行代码。以太坊采用状态 Patricia 树进行所有账户信息的表示和存储，状态根节点的哈希值如果相同，则说明所有联网节点都具有相同的账户状态。

#### （2）驱动交易

以太坊的交易驱动状态机运行，可能是简单的账户转账交易，也可能是合约账户的执行交易，或合约账户的创建交易。每一个交易的执行都会改变全局的状态 Patricia 树上的节



点或节点值,从而导致状态根节点的哈希值发生变化。

### (3) 转换引擎

一个状态机的运行需要一个引擎接收驱动交易并执行,根据交易结果调整状态 Patricia 树。如果是合约账户,合约交易的执行会把合约的二进制代码放在 EVM 上运行以达到最大的安全隔离。EVM 的执行结果,如合约各种状态存储值的变化,合约执行后相关账户余额的变化,都会反映到新的状态 Patricia 树上。

### (4) 通信服务

由于是分布式的应用,需要在各个节点之间进行 P2P 通信,确保节点可以动态组网、进行区块的同步、挖矿节点广播等,针对特定的使用目的会制定一些通信协议,如 Whisper 协议。另外,节点除了作为网络成员的角色外,还需要接受本地分布式 App 或 CLI 的接口调用,如支持本地节点的各种查询(如交易查询、余额查询等)、支持各种交易的发起(如执行合约的部署、合约的执行、账户的转账等),所以还需要提供 RPC/IPC 的客户端调用接口。

### (5) 共识验证

区块链的核心功能是可以分布在分布式的节点上实现共识,即对一组状态达成一致。以太坊目前沿用比特币的 PoW 挖矿算法,但是由于是全网共识,交易执行效率还是非常低下,并且需要消耗大量的能源,不利于大规模商业应用。基于此,以太坊正在向非 PoW 共识方面调整,转向 PoS 的 Casper 进行模拟挖矿。

## 8.3 以太坊智能合约

以太坊之所以被称为图灵完备的公有链,就在于其提供了可以在 EVM 上运行的智能合约机制。把智能合约中所有的状态数据维护在合约账户地址的存储空间上,每次向合约发送交易都会带来智能合约中状态数据的变化,这些状态数据会因为共识达成而在所有的客户端节点上保持一致。

由于是图灵完备的,智能合约可以用来建模现实世界各种业务、组织行为和规则,智能合约之间的交互又可以进一步影响现实世界中各类实体的交互行为,包括各种资产所有权的转移、数字资产和货币的支付等。

### 8.3.1 合约类型和调用示例

首先举一个多账户实体间交互的示例(见图 8-7)。这个示例是一个对赌协议,类似的对赌协议会出现在现实中的各种金融服务和产品中,如风投、保险条款、期货等。

Bob 和 Alice 对赌在未来的某天某地最高温度不超过 40°C, Bob 和 Alice 各预先存放了 50 代币在 GavCoin 处保管。如果当地当天温度超过 40°C, GavCoin 付 100 代币给 Bob,反之 GavCoin 付 100 代币给 Alice。

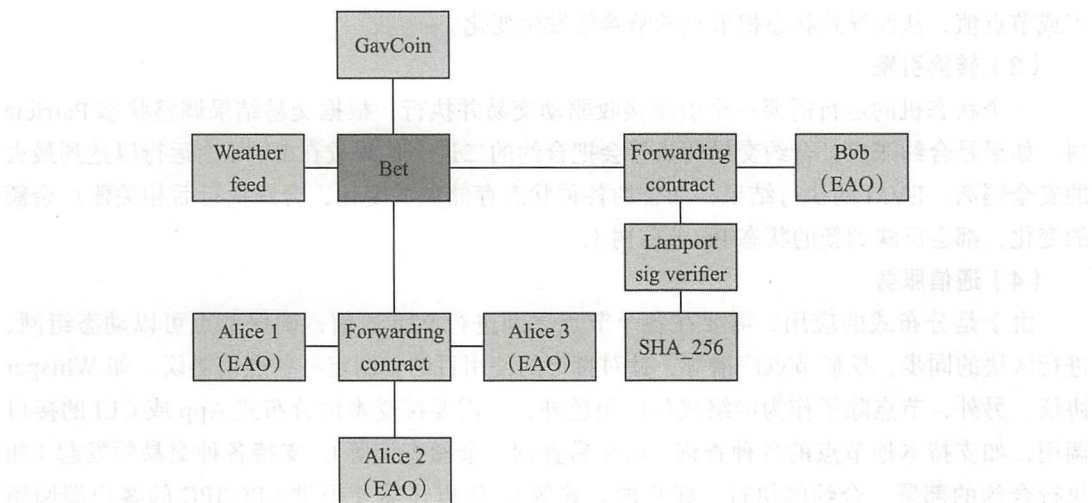


图 8-7 以太坊合约账户类型和交互示例

以太坊的合约可以有 4 种类型。

1) 维护对其他合约或外部世界有用的信息，如代币铸币，以及这里充当 Oracle（预言机）的 Weather feed（获取某地某天的最高温度）。

2) 执行复杂的外部账户访问策略，称为 Forwarding contract，如执行多重签名控制、钱包账户的取现限额控制。此处 Alice 比较谨慎，需通过大于 2 个私钥签名才发送消息给赌约 Bet<sup>⊖</sup>，Bob 需通过 Lamport 签名验证才可发消息给赌约 Bet。

3) 管理多人参与的、具有业务流程的合约和人员关系，如金融合约；存在特定中间人的第三方保管、悬赏合约，此处如 Bet 赌约。GavCoin 代为保管 Alice 和 Bob 的赌约筹码。

4) 作为合约代码库提供给其他合约调用使用，如此处的 Lamport sig verifier、SHA-256。

Bob 发起终止赌约，整个合约的调用关系如图 8-8 所示。

1) Bob 发送带有 Lamport 签名的消息给 Forwarding contract。

2) Forwarding contract 调用代码库 Lamport sig verifier，进行 Lamport 签名验证，Lamport sig verifier 又调用了 SHA-256 代码库，如果验证通过，Forwarding contract 发送消息给赌约 Bet。

3) 赌约 Bet 通过 Weather feed 获得某地某天的最高温度，发现高于 40°C。

4) 赌约 Bet 发消息给 GavCoin 通知其将 100 代币转至 Bob 名下。

用户可以使用以太坊提供的高级语言编写智能合约，编译成 EVM 字节码后部署到区块链上。目前支持 Solidity（类 JavaScript），Serpent（类 Python，需基于 LLL 语言进行编译），LLL（类 Lisp）三种高级语言，其中以 Solidity 最为流行。另外还有一个类 C 的 Mutan 语言已

⊖ 执行赌约的主智能合约程序。



经废弃不再维护。多种合约高级语言支持遵循以太坊多元化的思想，就如同提供了多种语言版本的以太坊客户端一样。

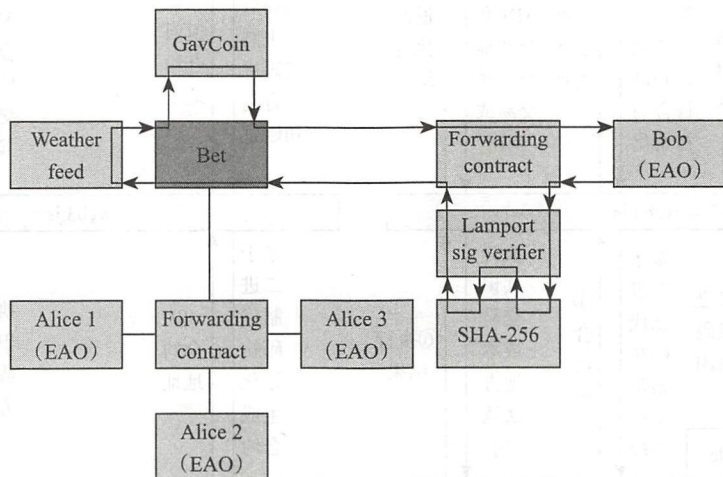


图 8-8 以太坊合约账户交互过程示例

### 8.3.2 合约编译和部署过程

合约部署到以太坊网络前需要把源码编译成 EVM 的二进制码，这里以 Solidity 合约语言编译和部署过程为例进行说明。图 8-9 为以太坊合约账户交互过程示例。

图 8-9 将以太坊智能合约开发测试和生产部署分开，在私有网络或者以太坊测试网络 (testnet) 执行开发测试，在以太坊网络 (Ethereum) 执行正式的生产部署。

#### (1) 合约编译和测试

Solidity 合约在合约编辑器或集成开发环境开发后，可以通过 web3.js 执行合约的编译和测试。web3.js 是以太坊提供的对外 API 接口，web3.js 会发起对以太坊节点上编译器 solc 的调用，对源码进行编译，或者采用命令行的方式直接使用 solc 命令对源码进行编译。编译和测试的过程主要包括以下几步。

- ① 发送源码：将合约 Solidity 源码通过 web3.js 发送给 Solidity 编译器 solc 执行编译。
- ② 二进制代码和 ABI：编译器生成的二进制代码和 ABI 返回给客户端脚本。
- ③ 部署编译后二进制代码进行合约初始化：将二进制代码和初始化的值作为参数发起合约实例的部署，形成一个交易，挖矿节点会将合约初始化交易打包进区块。
- ④ 合约地址：通过交易哈希获取交易收据，收据中会返回交易对应合约的地址，合约地址返回给客户端脚本。
- ⑤ 基于 ABI 和合约地址发送交易或调用：部署节点或者其他节点就可以基于合约 ABI 和合约地址 at 出一个合约调用 stub (存根)，通过这个 stub 执行交易发送或者方法调用 (call)。

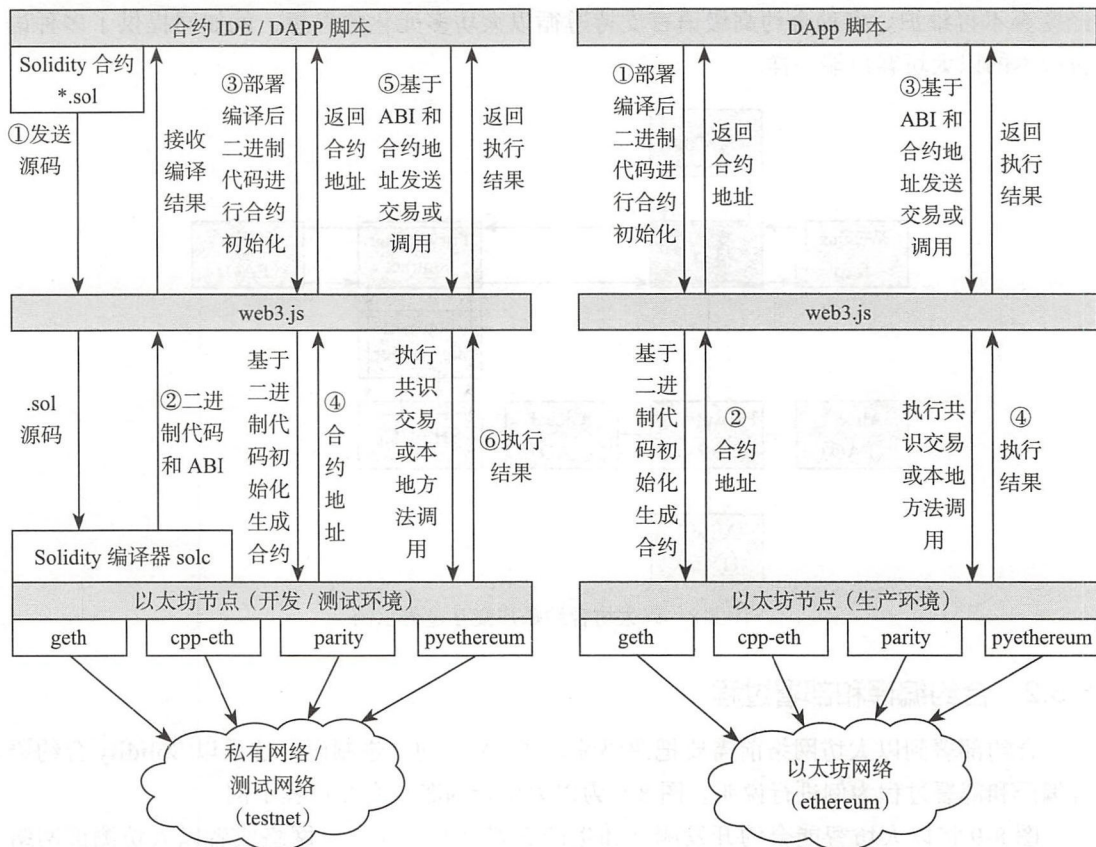


图 8-9 以太坊合约账户交互过程示例

⑥ 执行结果：交易发送是异步的，交易发送结果可以通过交易哈希基于交易收据获得，交易共识后的结果为每个节点共享，由于通过交易的方式需要打包进区块形成共识，所以需要消耗 Ether；对于无须修改区块链存储的方法，可以直接通过方法调用（call），只在本地节点执行 EVM 方法调用，调用是同步的，且无须消耗 Ether。

## （2）合约正式环境部署和使用

测试好的合约，可以使用编译后的二进制代码在以太坊正式网络上执行部署，部署后，全世界的用户都可以访问和使用部署的合约。部署和访问的过程同合约编译和测试步骤③~⑥，这里不再描述。

以太坊网络由基于黄皮书的若干种语言实现客户端版本组成，如基于 Go 语言的 geth、基于 C++ 语言的 eth、基于 Rust 语言的 parity、基于 Python 语言的 pyethereum，还有基于 Java 语言的 ethereumj。各种语言版本的实现一方面可以保证网络不会因为一种实现版本的 Bug 出现分叉，另一方面也保证了不同的实现方可以在语义层面（以太坊黄皮书）达成共识，形成更广泛的分布性。



### 8.3.3 Solidity 高级合约语言

Solidity 是一个语法类似于 JavaScript 的、面向合约的高级语言，编译后的二进制在 EVM 中运行。Solidity 是一个静态类型语言，支持继承、库访问、复杂的用户定义类型等特性。Solidity 高级合约语言目前在以太坊的合约中使用最为广泛，本节将讲述该语言的基本构成。

#### 1. 一个简单代币合约示例

图灵完备的智能合约的强大之处在于，用户可以使用高级编程语言实现自定义的公链链上资产，并进行链上资产相关的业务操作。以智能合约最简单的应用——代币发行为例：它基于比特币的脚本语言难以实现，然而在以太坊上却可以通过简单的几行代码实现。

图 8-10 实现了一个简单的代币合约。该合约包括铸币发行和代币转让。

```
pragma solidity ^0.4.0; <————— 声明solidity版本

contract Coin {
    // The keyword "public" makes those variables
    // readable from outside.
    address public minter; <————— public 声明可以使其他合约访问到这个数据状态
    mapping (address => uint) public balances; <————— mapping是更复杂的数据类型，维护一组键值对数据状态

    // Events allow light clients to react on
    // changes efficiently.
    event Sent(address from, address to, uint amount);

    // This is the constructor whose code is
    // run only when the contract is created.
    function Coin() {
        minter = msg.sender; <————— 合约初始化时，设定合约的创建者为铸币者
    }

    function mint(address receiver, uint amount) { <————— 代币铸币发行
        if (msg.sender != minter) return; <————— 只有合约的建立者才可以铸币
        balances[receiver] += amount;
    }

    function send(address receiver, uint amount) { <————— 代币转账发行
        if (balances[msg.sender] < amount) return;
        balances[msg.sender] -= amount; <————— 只要余额充足，任何发送者都可发起send交易，转账给他人
        balances[receiver] += amount;
        Sent(msg.sender, receiver, amount); <————— 发送event通知，转账成功。系统通过日志记录，并用最多3个参数
        // 进行索引，提供给本合约事件监听者
    }
}
```

图 8-10 智能合约代码示例

第一行“pragma solidity ^0.4.0;”声明合约源码适用的 Solidity 编译器版本，^0.4.0 代表此源码使用 Solidity 0.4.0 或以上的版本编写，但是不适用包括 0.5.0 及以上的版本，这可以确保合约在相对固定的编译器版本上实现编译的稳定性。

“address public minter;”声明一个地址类型的状态变量，地址是 20 个字节的账户地址，public 关键词会自动生成一个对状态变量的外部访问方法，其他合约就可以访问这个状态变量。



“mapping (address => uint) public balances;” 中 mapping 是一个更加复杂的类型，维护一组键值对。由于无法枚举或循环获取 key 或 value，使用时需要记录加入 mapping<sup>⊖</sup> 的 key，并使用 key 获取对应的值。

“event Sent (address from, address to, uint amount);” 由交易执行的方法触发，轻客户端可以监听事件，当事件发生时，进行回调处理。例如，事件监听和回调的处理逻辑如下：

```
Coin.Sent().watch({}, '', function(error, result) {
    if (!error) {
        console.log("Coin transfer: " + result.args.amount +
            " coins were sent from " + result.args.from +
            " to " + result.args.to + ".");
        console.log("Balances now:\n" +
            "Sender: " + Coin.balances.call(result.args.from) +
            "Receiver: " + Coin.balances.call(result.args.to));
    }
})
```

说明：

1) function Coin() 是合约的构造器函数，构造器函数在合约创建时执行，创建者发送合约创建交易，并设定构造器的参数。此处，构造器的逻辑是把合约创建者地址作为铸币者，只有铸币者才可以铸造代币。此处的 msg (还有 tx 和 block) 是内置的全局对象变量，可以通过对象的属性访问区块链相关的值。

2) function mint() 可以接收由合约创建者发起的铸币交易，如果不是合约创建者发起的 mint 交易，则忽略。

3) function send() 可以由任何账户发起调用，只要其拥有的余额足够支付交易中指定的金额。

4) Sent (msg.sender, receiver, amount); 可以在执行完转账后触发一个 event 通知，所有监听这个 event 的客户端都可以及时得到事件通知，并执行相关的回调函数。

## 2. Solidity 编译器的安装

### (1) Remix

这是一个集成了 Solidity 编译器的、基于浏览器的合约 IDE (见图 8-11)，用户可以通过浏览器进行合约的编写、编译和发布测试。Remix 的访问地址是：<https://remix.ethereum.org/>。

<sup>⊖</sup> mapping 是一种数据结构，类似 hashmap、dictionary。





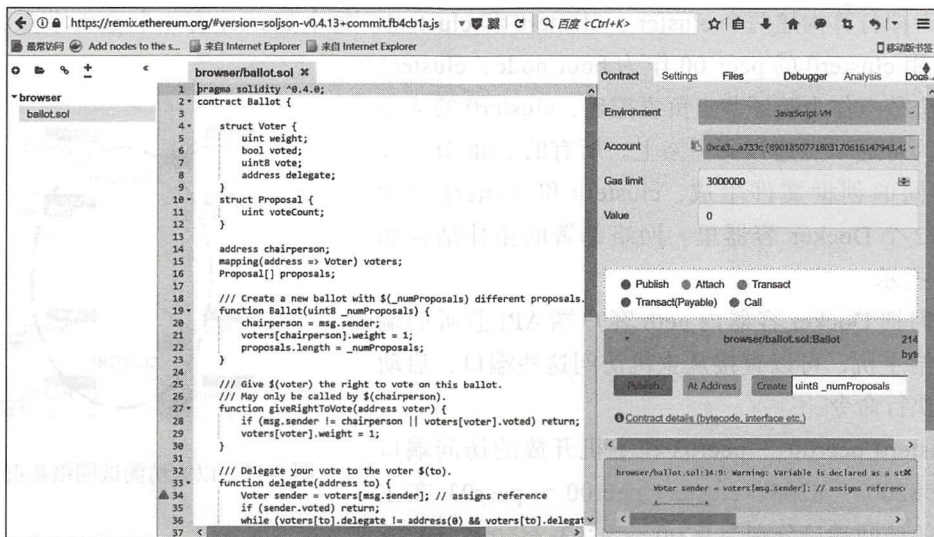


图 8-11 Remix 在线合约开发 IDE 界面

## (2) npm/Node.js 版本

由于 Solidity 是 C++ 编写，使用 Emscripten 将 C++ 源码编译成 JavaScript 库 solc-js。这个 JavaScript 库目前也直接集成在 Remix 中使用。JavaScript 版本还提供命令行 solcjs，可以通过 npm 命令进行安装：npm install -g solc。JavaScript 版本的具体使用可以参考：<https://github.com/ethereum/solc-js>。

## (3) Docker 版本

Solidity 编译后的版本已经发布在 Docker 中，可以使用 Docker 运行容器并执行 solc 命令，进行合约的编译：

```
docker run ethereum/solc:stable solc --version # 查看 solc 当前版本
```

笔者也制作了一个 Docker 版本：blockchain101/ethereum-solc:0.4.11，在后续的案例中会使用到。由于篇幅所限，二进制的安装和从源码进行编译请参考：<http://solidity.readthedocs.io/en/latest/installing-solidity.html>。Solidity 具体语法请参考：<http://solidity.readthedocs.io/en/latest/solidity-in-depth.html>。

## 8.3.4 案例：构建、编译与部署一个智能合约

### 1. 以太坊测试网络的构建说明

为了快速构建以太坊的多节点集群网络环境，笔者将以太坊启动参数化，通过 Docker compose 方式按照需要启动 cluster 服务。本案例<sup>①</sup>以图 8-12 为例构建区块链网络。

① 参见《Docker 容器化快速构建多集群以太坊网络并部署智能合约》，<http://bctrustmachine.cn/forum.php?mod=viewthread&tid=59&extra=page%3D1>。



笔者拟打算构建两个 cluster：cluster0 和 cluster1。每个 cluster 有 4 个 geth 节点，这些节点都用 cluster0 的 peer 00 作为 boot node，cluster1 上的 4 个节点作为验证节点负责挖矿，cluster0 的 4 个节点作为普通节点连接到网络上。所有的 geth 节点都使用相同的创世文件生成，cluster0 和 cluster1 分别部署在 2 个 Docker 容器里。网络部署的拓扑结构如图 8-11 所示。

我们把 Docker 容器内 geth 客户端 API 监听的端口暴露给主机，可以直接从主机访问这些端口，启动命令行执行命令。

cluster0 peer00 ~ peer03 在主机开放的访问端口分别为 8200 ~ 8203，cluster1 peer00 ~ peer03 在主机开放的访问端口分别为 9200 ~ 9203。

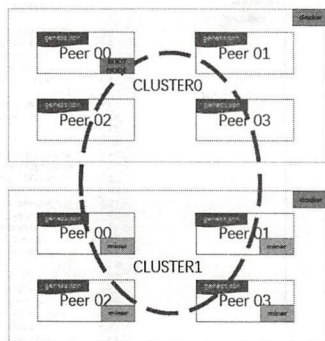


图 8-12 以太坊测试网络集群架构图



按照实际情况，读者可以不必拘泥于上述结构，可以部署更多的 cluster，每个 cluster 也可以只有 1 个 geth 节点，只需按照要求修改 docker compose 文件即可。

## 2. 测试环境构建过程

### (1) 环境准备

本次实践我们部署的 Docker 都运行在一个 Ubuntu Linux 上，其主要参数如下：

- ❑ 操作系统：Ubuntu 16.04 LTS
- ❑ 内存：不小于 4GB
- ❑ IP: 192.168.5.172
- ❑ gateway: 192.168.5.2
- ❑ hostname: ubu-blockchain2
- ❑ Docker Network Bridge gateway: 172.16.238.0.1

由于指定 IP，运行的两个 cluster 所在 container 的 IP 如下：

- ❑ 172.16.238.10: geth-cluster0;
- ❑ 172.16.238.11: geth-cluster1。

请安装 Git 用于从 GitHub 下载资源。Git 的安装请参考：<https://git-scm.com/download/linux>。

请根据自身的环境安装最新版本的 Docker 及 docker-compose。Docker 的安装请参考：<https://docs.docker.com/engine/installation/>。docker-compose 的安装请参考：<https://docs.docker.com/compose/install/>。



由于是在 Docker 容器内运行，读者也可以选用其他的 OS 环境，如在 Windows 环境或其他 Linux 版本下进行部署和运行。





## (2) 下载最新的 ethereum-docker 资源

执行命令：

```
$ git clone https://github.com/blockchain101/ethereum-docker.git
```

执行结果如图 8-13 所示。

```
rodger@ubu-blockchain2:~$ git clone https://github.com/blockchain101/ethereum-docker.git
Cloning into 'ethereum-docker'...
remote: Counting objects: 15, done.
remote: Compressing objects: 100% (13/13), done.
remote: Total 15 (delta 1), reused 14 (delta 0), pack-reused 0
Unpacking objects: 100% (15/15), done.
Checking connectivity... done.
```

图 8-13 执行结果

## (3) 启动测试网络

由于是内部测试网络，在启动网络前最好先断开外网，这样就可以屏蔽掉外网的 Ethereum 节点连接尝试，去除日志干扰。

执行命令：

```
$ cd ethereum-docker/ethereum-docker/ethereum-testnet-docker/
$ docker-compose -f docker-compose-ethereum-testnet.yaml up
```

由于 docker-compose 文件中指定了本地不存在的 image blockchain101/ethereum-geth:1.6.5，Docker 会自动从 GitHub 下载指定的 image，如图 8-14 所示。

```
rodger@ubu-blockchain2:~$ cd ethereum-docker/ethereum-docker/ethereum-testnet-docker/dockercomposefiles/
rodger@ubu-blockchain2:~/ethereum-docker/ethereum-docker/ethereum-testnet-docker/dockercomposefiles$ docker-compose -f docker-compose-ethereum-testnet.yaml up
WARNING: The Docker Engine you're using is running in swarm mode.
Compose does not use swarm mode to deploy services to multiple nodes in a swarm. All containers will be scheduled on the current node.
to deploy your application across the swarm, use 'docker stack deploy'.
Creating network "dockercomposefiles_etherneumnet" with driver "bridge"
Pulling geth-cluster0 (blockchain101/ethereum-geth:1.6.5)...
1.6.5: Pulling from blockchain101/ethereum-geth
8f995eb2a3: Already exists
8f975b472e7: Already exists
e931b117db38: Already exists
47b5161e0b11: Already exists
9332ea11a539: Already exists
b29ea12800bd: Already exists
8607c04a9f4c: Already exists
dab17b17a798: Already exists
8c7a843ff5e6: Already exists
1ef8069b789a: Already exists
d572ce0e9942: Already exists
76b67768f258: Already exists
8ce93584c8a2: Already exists
151a08083204: Already exists
423b1203f6fd: Already exists
3543463b33ba: Already exists
05909979191: Already exists
da971dbfae2c: Already exists
9520952b65df: Already exists
74d2a8af60f4: Already exists
4f7e533ae23c: Already exists
53f4e14d6436: Already exists
1ee265380618: Already exists
e14c45c38bd0: Already exists
420a4cha61c3: Pull complete
613e490c39e9: Pull complete
0e07e806c51f: Downloading [====>] 3.36MB/3.635MB
4da85ccc3381: Download complete
5t27410858de: Downloading [====>] 1.596MB/89.81MB
36a0d1dba78d: Download complete
85af8da906c2: Download complete
47da21bf44bf: waiting
4dc0d4790762: waiting
e9bdc941a86f: waiting
8c7e232eb7e8: waiting
69345d150722: waiting
da13a9e9f157: waiting
e80e1a5dd3b3: waiting
```

图 8-14 启动测试网络下载 image

下载完成后按照配置启动两个容器 geth-cluster0 和 geth-cluster1，每个容器运行 4 个 geth 客户端节点。图 8-15 所示是 cluster0 和 cluster1 的启动过程界面。





```

geth-cluster1 DEBUG [06-14 02:30:38] Relaxed downloader QoS values
geth-cluster1 DEBUG [06-14 02:30:38] Adding p2p peer
geth-cluster1 DEBUG [06-14 02:30:38] Ethereum peer connected
geth-cluster1 DEBUG [06-14 02:30:38] Relaxed downloader QoS values
geth-cluster1 DEBUG [06-14 02:30:38] Adding p2p peer
geth-cluster1 DEBUG [06-14 02:30:38] Ethereum peer connected
geth-cluster1 DEBUG [06-14 02:30:38] Relaxed downloader QoS values
geth-cluster1 DEBUG [06-14 02:30:38] Adding p2p peer
geth-cluster1 DEBUG [06-14 02:30:38] Ethereum peer connected
geth-cluster1 DEBUG [06-14 02:30:38] Relaxed downloader QoS values
geth-cluster1 DEBUG [06-14 02:30:38] Removing p2p peer
geth-cluster1 DEBUG [06-14 02:30:38] Adding p2p peer
geth-cluster1 DEBUG [06-14 02:30:38] Ethereum peer connected
geth-cluster1 DEBUG [06-14 02:30:38] Relaxed downloader QoS values
geth-cluster1 INFO [06-14 02:30:41] Generating ethash verification cache
geth-cluster1 INFO [06-14 02:30:41] Generated ethash verification cache
geth-cluster1 DEBUG [06-14 02:30:52] Recalculated downloader QoS values
geth-cluster1 DEBUG [06-14 02:30:52] Recalculated downloader QoS values
geth-cluster1 INFO [06-14 02:31:09] Generating DAG in progress
geth-cluster1 DEBUG [06-14 02:31:12] Recalculated downloader QoS values
geth-cluster1 DEBUG [06-14 02:31:13] Recalculated downloader QoS values
geth-cluster1 DEBUG [06-14 02:31:32] Recalculated downloader QoS values
geth-cluster1 DEBUG [06-14 02:31:32] Recalculated downloader QoS values
geth-cluster1 INFO [06-14 02:31:52] Generating DAG in progress
geth-cluster1 DEBUG [06-14 02:31:52] Recalculated downloader QoS values
geth-cluster1 DEBUG [06-14 02:31:52] Recalculated downloader QoS values
geth-cluster1 INFO [06-14 02:32:05] Generating DAG in progress
geth-cluster1 DEBUG [06-14 02:32:12] Recalculated downloader QoS values
geth-cluster1 DEBUG [06-14 02:32:13] Recalculated downloader QoS values
geth-cluster1 DEBUG [06-14 02:32:32] Recalculated downloader QoS values
geth-cluster1 INFO [06-14 02:32:32] Generating DAG in progress
geth-cluster1 DEBUG [06-14 02:32:33] Recalculated downloader QoS values
geth-cluster1 DEBUG [06-14 02:32:52] Recalculated downloader QoS values
geth-cluster1 DEBUG [06-14 02:32:52] Recalculated downloader QoS values
geth-cluster1 INFO [06-14 02:33:01] Generating DAG in progress
geth-cluster1 DEBUG [06-14 02:33:12] Recalculated downloader QoS values
geth-cluster1 DEBUG [06-14 02:33:13] Recalculated downloader QoS values
geth-cluster1 INFO [06-14 02:33:29] Generating DAG in progress
geth-cluster1 DEBUG [06-14 02:33:32] Recalculated downloader QoS values
geth-cluster1 DEBUG [06-14 02:33:33] Recalculated downloader QoS values
geth-cluster1 DEBUG [06-14 02:33:52] Recalculated downloader QoS values
geth-cluster1 DEBUG [06-14 02:33:52] Recalculated downloader QoS values
geth-cluster1 INFO [06-14 02:33:58] Generating DAG in progress
geth-cluster1 DEBUG [06-14 02:34:12] Recalculated downloader QoS values
geth-cluster1 DEBUG [06-14 02:34:13] Recalculated downloader QoS values
geth-cluster1 INFO [06-14 02:34:33] Generating DAG in progress
geth-cluster1 DEBUG [06-14 02:34:33] Recalculated downloader QoS values
geth-cluster1 DEBUG [06-14 02:34:33] Recalculated downloader QoS values
geth-cluster1 DEBUG [06-14 02:34:52] Recalculated downloader QoS values
geth-cluster1 DEBUG [06-14 02:34:52] Recalculated downloader QoS values
geth-cluster1 INFO [06-14 02:34:57] Generating DAG in progress
geth-cluster1 DEBUG [06-14 02:35:12] Recalculated downloader QoS values
geth-cluster1 DEBUG [06-14 02:35:13] Recalculated downloader QoS values
geth-cluster1 INFO [06-14 02:35:26] Generating DAG in progress
geth-cluster1 DEBUG [06-14 02:35:32] Recalculated downloader QoS values
geth-cluster1 DEBUG [06-14 02:35:33] Recalculated downloader QoS values
geth-cluster1 DEBUG [06-14 02:35:52] Recalculated downloader QoS values
geth-cluster1 DEBUG [06-14 02:35:52] Recalculated downloader QoS values
geth-cluster1 INFO [06-14 02:35:55] Generating DAG in progress

```

图 8-15 cluster0 和 cluster1 的启动过程

由于初始值需要做全局初始化，生成各节点账号，生成节点数据文件，如果指定基于 genesis 文件，会生成基于 genesis 文件的区块链创世区块，另外整个集群会设定某个节点作为 bootnode。当然为了可用性，可以改进为让每个 cluster 都设定一个节点作为 bootnode。默认的配置 cluster0 中的 4 个节点是观察节点，cluster1 的 4 个节点是验证挖矿节点。bootnode 文件、genesis 文件以及 ethash DAG 文件作为所有集群共享的文件放在 ethcluster\_share 目录中。系统还自动创建出 ethcluster 目录，用于放置 cluster0 和 cluster1 的区块链数据文件。

在当前目录会自动生成 ethcluster、ethcluster\_share 目录，如图 8-16 所示。

```

rodger@ubuntu:~/blockchain2:~/etherum-docker/ethereum-docker/ethereum-testnet-docker/dockercomposefiles$ ls -ltr
total 20
-rw-rw-r-- 1 rodger rodger 1974 Jun 13 03:16 README.md
-rw-rw-r-- 1 rodger rodger 912 Jun 13 03:16 genesis-779977.json
drwxr-xr-x 4 root root 4096 Jun 13 19:30 ethcluster
drwxr-xr-x 3 root root 4096 Jun 13 19:30 ethcluster_share
-rw-rw-r-- 1 rodger rodger 3344 Jun 13 19:35 docker-compose-ethereum-testnet.yaml

```

图 8-16 自动生成的 ethcluster 与 ethcluster\_share 目录

由于是第一次启动挖矿节点，需要建立 DAG。这个过程会花费很长时间，在这个过程中，我们先做一些配置修改，由于 cluster0 的 4 个节点都不挖矿，我们的目的是借助于 genesis 文件赋予 cluster0 的 4 个节点的账号有初始的 ether，分别初始化 10、11、12、13 个 ether 给这 4 个节点，这就需要把这 4 个节点的账号设置到 genesis 文件中。请按照图 8-17 先查看 cluster04 个节点对应的账号。

执行命令：

```
sudo ls -lR ethcluster/cluster0/779977/keystore
```





```

rodger@ubu-blockchain2:~/ethereum-docker/ethereum-docker/ethereum-testnet-docker/dockercomposefiles$ sudo ls -lR ethcluster/cluster0/779977/keystore/
ethcluster/cluster0/779977/keystore/:
total 16
drwxr-xr-x 3 root root 4096 Jun 13 19:30 00
drwxr-xr-x 3 root root 4096 Jun 13 19:30 01
drwxr-xr-x 3 root root 4096 Jun 13 19:30 02
drwxr-xr-x 3 root root 4096 Jun 13 19:30 03

ethcluster/cluster0/779977/keystore/00:
total 4
drwx----- 2 root root 4096 Jun 13 19:30 keystore

ethcluster/cluster0/779977/keystore/00/keystore:
total 4
-rw----- 1 root root 491 Jun 13 19:30 UTC--2017-06-14T02-30-00.426014740Z--6a9cc5466f3274efabaac7907895e1ebf49911a7

ethcluster/cluster0/779977/keystore/01:
total 4
drwx----- 2 root root 4096 Jun 13 19:30 keystore

ethcluster/cluster0/779977/keystore/01/keystore:
total 4
-rw----- 1 root root 491 Jun 13 19:30 UTC--2017-06-14T02-30-01.391243188Z--c7f5a898d14f91fde9a3d37c150c6a4f9b104545

ethcluster/cluster0/779977/keystore/02:
total 4
drwx----- 2 root root 4096 Jun 13 19:30 keystore

ethcluster/cluster0/779977/keystore/02/keystore:
total 4
-rw----- 1 root root 491 Jun 13 19:30 UTC--2017-06-14T02-30-02.353888907Z--939957062d814583e37ec34a0a159f2e8cb7a3d0

ethcluster/cluster0/779977/keystore/03:
total 4
drwx----- 2 root root 4096 Jun 13 19:30 keystore

ethcluster/cluster0/779977/keystore/03/keystore:
total 4
-rw----- 1 root root 491 Jun 13 19:30 UTC--2017-06-14T02-30-03.389143058Z--6522264929aa4ab0e3905dbb441a540c042770c4
rodger@ubu-blockchain2:~/ethereum-docker/ethereum-docker/ethereum-testnet-docker/dockercomposefiles$

```

图 8-17 查看自动生成的节点账户

图 8-17 中每个节点的私钥文件名（UTC- 开头的文件）的后 20 个字符就是自动生成的账户地址，然后用上述的账户地址更新 genesis 文件中的 alloc 部分的账号内容，将这 4 个账户地址替换 genesis-779977.json 文件 alloc 中的 4 个地址，确保 cluster0 的这 4 个账户地址一一对应，如图 8-18 所示。

```

rodger@ubu-blockchain2:~/ethereum-docker/ethereum-docker/ethereum-testnet-docker/dockercomposefiles$ cat genesis-779977.json
{
  "config": {
    "chainId": 779977,
    "homesteadBlock": 0,
    "eip155Block": 0,
    "eip158Block": 0
  },
  "nonce": "0x779977779977799",
  "timestamp": "0x00",
  "parentHash": "0x0000000000000000000000000000000000000000000000000000000000000000",
  "extradata": "0x00",
  "gasLimit": "0x8000000",
  "difficulty": "0x400",
  "mixhash": "0x0000000000000000000000000000000000000000000000000000000000000000",
  "coinbase": "0x3333333333333333333333333333333333333333333333333333333333333333",
  "alloc": {
    "0x6a9cc5466f3274efabaac7907895e1ebf49911a7": {
      "balance": "1000000000000000000000000"
    },
    "0xc7f9a898d14f91fde9a3d37c150c6a4f9b104545": {
      "balance": "1100000000000000000000000"
    },
    "0x939957062d814583e37ec34a0a159f2e8cb7a3d0": {
      "balance": "1200000000000000000000000"
    },
    "0x6522264929aa4ab0e3905dbb441a540c042770c4": {
      "balance": "1300000000000000000000000"
    }
  }
}

```

图 8-18 修改创世文件替换地址

我们需要重新设定 docker compose 的启动选项，将 cluster0 的 INIT\_CONFIG 从 Y 修改成 GENESIS\_BLOCK\_REGEN。重新启动 Docker 后，cluster0 不会再生成账号，但是会重建区块链 genesis block 数据，cluster1 还是会重新生成账号和基于新的 genesis 文件初始化 genesis block 并启动。这时这个网络都会认同上述对 genesis 文件的修改，确保 cluster0 的 4 个节点分别有 10、11、12、13 个初始的 ether 值。当然这个过程可以完全利用脚本自





动化一次性替换实现，但是为了向初学者演示以太坊在创世文件设定账户初始值的这个特性，还是采用手动的方式进行替换操作。

执行命令：

```
vi docker-compose-ethereum-testnet.yaml
```

将 geth0-cluster0 的 INIT\_CONFIG 的值从 Y 修改成 GENESIS\_BLOCK\_REGEN，如图 8-19 所示，保存和退出。

```
geth0-cluster0:
  container_name: geth-cluster0
  image: blockchain101/ethereum-geth:1.6.5
  environment:
    - CLUSTER_DATA_DIR=/ethcluster
    - NETWORK_ID=779977
    - CLUSTER_INSTANCE_NUM=1
    - CLUSTER_BOOT_NODE=Y
    - CLUSTER_MINING=N
    - CLUSTER_ID=0
    - CLUSTER_SHAREDIR=/ethcluster_share
  #####INIT_CONFIG SETTING#####
  # Y - recreate accounts, nodes and nodes data with/without genesis file based on GENESIS_FILE_ENABLED flag
  # N - no accounts, nodes and nodes data regeneration if not the first time initialization nodes and accounts
  # GENESIS_BLOCK_REGEN - no accounts regeneration, but nodes and nodes data will be regenerated with/without genesis file based on GENESIS_FILE_ENABLED flag
  # INIT_CONFIG=GENESIS_BLOCK_REGEN
  # GENESIS_FILE_ENABLED=Y
  # GENESIS_FILE=/ethcluster_share/genesis-779977.json
  command: bash -c " /rungeth.sh; tail -f /ethcluster/779977/log/00.log"
```

图 8-19 修改 INIT\_CONFIG 的值

如果屏幕还是看到如图 8-15 所示的日志效果，说明目前挖矿节点还在生成 epoch0 前 30 000 个区块。挖矿要使用的 DAG，生成的 DAG 文件存放在 ethcluster\_share/ethash 目录中。

我们继续等待第一次启动的 geth-cluster1 挖矿节点把 DAG 都生成好。直到 geth-cluster1 出现 mining potential block 日志提示，如图 8-20 所示，说明 DAG 已经准备好 30 000 blocks epoch0，开始执行挖矿了。

```
geth-cluster1 DEB[06-14 11:50:58] Recalculated downloader QoS values
geth-cluster1 INFO [06-14 11:50:59] Generating DAG in progress
geth-cluster0 DEB[06-14 11:51:18] Recalculated downloader QoS values
geth-cluster0 DEB[06-14 11:51:19] Recalculated downloader QoS values
geth-cluster1 DEB[06-14 11:51:19] Recalculated downloader QoS values
geth-cluster0 DEB[06-14 11:51:19] Recalculated downloader QoS values
geth-cluster1 DEB[06-14 11:51:19] Recalculated downloader QoS values
geth-cluster0 DEB[06-14 11:51:19] Recalculated downloader QoS values
geth-cluster1 DEB[06-14 11:51:19] Recalculated downloader QoS values
geth-cluster1 INFO [06-14 11:52:00] Generating DAG in progress
geth-cluster1 DEB[06-14 11:52:19] Recalculated downloader QoS values
geth-cluster0 DEB[06-14 11:52:21] Successfully sealed new block
geth-cluster1 DEB[06-14 11:52:21] Tric cache stats after commit
geth-cluster1 INFO [06-14 11:52:21] mined potential block
geth-cluster1 INFO [06-14 11:52:21] Commit new mining sort
geth-cluster0 DEB[06-14 11:52:22] Quoted propagated block
geth-cluster0 DEB[06-14 11:52:22] Importing propagated block
geth-cluster0 WARN [06-14 11:52:22] Discarded bad propagated block
geth-cluster0 DEB[06-14 11:52:22] Quoted propagated block
geth-cluster0 DEB[06-14 11:52:22] Importing propagated block
geth-cluster0 DEB[06-14 11:52:22] Fetching single header
geth-cluster0 DEB[06-14 11:52:22] Quoted propagated block
geth-cluster1 DEB[06-14 11:52:22] Fetching single header
geth-cluster1 DEB[06-14 11:52:22] Quoted propagated block
geth-cluster1 DEB[06-14 11:52:22] Importing propagated block
geth-cluster1 DEB[06-14 11:52:22] Unknown parent of propagated block
geth-cluster1 DEB[06-14 11:52:28] Recalculated downloader QoS values
geth-cluster0 DEB[06-14 11:52:30] Recalculated downloader QoS values
geth-cluster0 DEB[06-14 11:52:46] Quoted propagated block
geth-cluster0 DEB[06-14 11:52:46] Importing propagated block
geth-cluster0 WARN [06-14 11:52:46] Discarded bad propagated block
geth-cluster1 DEB[06-14 11:52:47] Quoted propagated block
geth-cluster1 DEB[06-14 11:52:47] Importing propagated block
geth-cluster0 DEB[06-14 11:52:47] Discarded bad propagated block
geth-cluster1 WARN [06-14 11:52:47] Discarded bad propagated block
geth-cluster1 INFO [06-14 11:52:51] Successfully sealed new block
geth-cluster1 WARN [06-14 11:52:51] Tric cache stats after commit
geth-cluster1 INFO [06-14 11:52:51] mined potential block
geth-cluster1 INFO [06-14 11:52:51] Commit new mining sort
geth-cluster0 DEB[06-14 11:52:51] Block synchronisation started
geth-cluster0 DEB[06-14 11:52:51] Synchronising with the network
geth-cluster0 DEB[06-14 11:52:51] Retrieving remote chain height
geth-cluster0 DEB[06-14 11:52:51] Fetching batch of headers
geth-cluster0 DEB[06-14 11:52:51] Quoted propagated block
geth-cluster0 DEB[06-14 11:52:51] Remote header identified
geth-cluster0 DEB[06-14 11:52:51] Looking for common ancestor
geth-cluster0 DEB[06-14 11:52:51] Fetching batch of headers
geth-cluster0 DEB[06-14 11:52:51] Remote header identified
geth-cluster0 DEB[06-14 11:52:51] Fast syncing until pivot block
geth-cluster0 DEB[06-14 11:52:51] Downloading remote state data
geth-cluster0 DEB[06-14 11:52:51] Directing header downloads
geth-cluster0 DEB[06-14 11:52:51] Downloading block bodies
geth-cluster0 DEB[06-14 11:52:51] Downloading transaction receipts
rtt-20s confidence=1.000 ttl=1m0s
epoch=1 percentage=1 elapsed=6m2.905s
peer-30f8e42922b6880 number=1 hash=7b84da-b66e7a queued-1
peer-30f8e42922b6880 number=1 hash=7b84da-b66e7a
number=1 hash=7b84da-b66e7a
peer-8d01f0f0adce29d3 number=1 hash=7b84da-b66e7a queued-1
number=1 hash=7b84da-b66e7a
id=30f8e42922b6880 conn=dyndial hash=07d262_e0b015
peer=30f8e42922b6880 number=2 hash=07d262_e0b015 queued-1
16-df500322c2f2203 conn=dyndial hash=07d262_e0b015
peer=1500322c2f2203 number=2 hash=07d262_e0b015
peer=1500322c2f2203 number=2 hash=07d262_e0b015
peer=1500322c2f2203 number=2 hash=07d262_e0b015 parent=b64cf6-7215fd
rtt-20s confidence=1.000 ttl=1m0s
peer-8d01f0f0adce29d3 number=1 hash=0ad96-b245e9 queued-2
peer=8d01f0f0adce29d3 number=1 hash=0ad96-b245e9
number=1 hash=0ad96-b245e9
peer=ca6a3e5f5f29e number=1 hash=0ad96-b245e9 queued-1
peer=ca6a3e5f5f29e number=1 hash=0ad96-b245e9
peer=5d4d7c488bf29e number=1 hash=0ad96-b245e9 queued-2
peer=5d4d7c488bf29e number=1 hash=0ad96-b245e9
number=1 hash=0ad96-b245e9
number=2 hash=0719-fa566e
rtt-20s confidence=1.000 ttl=1m0s
number=2 hash=dc0719-fa566e
number=3 txs=0 uncles=0 elapsed=133.632ps
peer=74b702beab8f880 eths=63 head=7b84da-b66e7a td=132096 mode=fast
peer=74b702beab8f880 conn=inbound count=1 fromhash=7b84da-b66e7a skip=0 reverse=false
peer=74b702beab8f880 number=2 hash=dc0719-fa566e queued-2
peer=74b702beab8f880 number=1 hash=7b84da-b66e7a
peer=74b702beab8f880 local=0 remote=1
16-df500322c2f2203 conn=inbound count=1 fromhash=7b84da-b66e7a skip=15 reverse=false
pivot=0
peer=74b702beab8f880 number=0 hash=e935bf-23170b
pivot=0
peer=74b702beab8f880 origin=1
origin=1
```

图 8-20 测试网络启动的终端显示开始执行挖矿





我们先使用 Ctrl+C 停止当前 docker compose，然后基于之前刚刚修改过的 docker compose 文件重启 Docker 容器。这次启动将不会为 geth-cluster0 的 4 个节点重新生成账户，geth-cluster0 和 geth-cluster1 都使用更新过的 genesis 启动区块链，共同认可 geth-cluster0 的 4 个节点设定的初始 ether 余额这个事实。

另外在主机上打开两个终端，分别输入命令，启动连接到 geth-cluster0 peer01 和 geth-cluster1 peer01 的控制台，查看网络组网状态和账户余额情况。

```
$ geth attach http://192.168.5.172:8201 # 访问 cluster0 peer01 console
$ geth attach http://192.168.5.172:8201 # 访问 cluster1 peer01 console
```

注意，如果本地机器没有 geth 命令，可以启动容器来执行 geth 命令：

```
$ docker run -i blockchain101/ethereum-geth:1.6.5 geth attach
http://192.168.5.172:8201 # 访问 cluster0 peer01 console
$ docker run -i blockchain101/ethereum-geth:1.6.5 geth attach
http://192.168.5.172:9201 # 访问 cluster1 peer01 console
```

在 cluster0 peer01 上执行 net.peerCount 并检查主账号的 ether 值，如图 8-21 所示。

```
rodger@ubu-blockchain2:~/ethereum-docker/ethereum-docker/ethereum-testnet-docker/dockercomposefiles$ geth attach http://192.168.5.172:8201
Welcome to the Geth JavaScript console!

instance: Geth/01/v1.6.5-stable/linux-amd64/go1.8
coinbase: 0xc7f9a898d14f91fde9a3d37c150c6a4f9b104545
at block: 114 (Wed, 14 Jun 2017 05:33:22 PDT)
datadir: /ethcluster/779977/data/01
modules: admin:1.0 eth:1.0 net:1.0 rpc:1.0 web3:1.0

> net.peerCount
7
> primary=eth.accounts[0]
"0xc7f9a898d14f91fde9a3d37c150c6a4f9b104545"
> balance=web3.fromWei(eth.getBalance(primary),"ether")
11
>
```

图 8-21 连接 cluster0 peer01 节点的命令行查询主账号余额

两个 cluster，每个 cluster 4 个节点，所以图 8-21 显示 peerCount 是 7。而 eth.accounts[0] 节点默认主账号是写在 genesis 文件中的账号 “0xc7f9a898d14f91fde9a3d37c150c6a4f9b104545”。图 8-20 显示的余额 ether 值是 11，在 cluster1 peer01 的 console 查看该账号值也是 11，如图 8-22 所示。另外 eth.accounts[0] 作为挖矿奖励的默认账户，目前已经获得了挖矿奖励 ether。

```
rodger@ubu-blockchain2:~/ethereum-docker/ethereum-docker/ethereum-testnet-docker/dockercomposefiles/ethcluster/cluster1/779977$ geth attach http://192.168.5.172:9201
Welcome to the Geth JavaScript console!

instance: Geth/01/v1.6.5-stable/linux-amd64/go1.8
coinbase: 0xc84923fccc0bb261372d48b08fd50ee457c4ea
at block: 176 (Wed, 14 Jun 2017 05:42:42 +00)
datadir: /ethcluster/779977/data/01
modules: admin:1.0 eth:1.0 net:1.0 rpc:1.0 web3:1.0

> net.peerCount
7
> balance=web3.fromWei(eth.getBalance("0xc7f9a898d14f91fde9a3d37c150c6a4f9b104545"),"ether")
11
> balance=web3.fromWei(eth.getBalance(eth.accounts[0]),"ether")
266.40625
>
```

图 8-22 连接 cluster1 peer01 节点的命令行查询 cluster0 peer01 节点和本节点主账号余额

读者可以检查其他几个写在 genesis 文件中的账号的 ether 值，也可以尝试使用其他端口连接到其他的节点控制台进行操作。





以此执行下面的控制台命令进行合约的发布和调用。

```
> primary=eth.accounts[0]
> balance=web3.fromWei(eth.getBalance(primary),"ether")
> loadScript("./ethcontract/Sample.abi")
> loadScript("./ethcontract/Sample.bin")
> sample=eth.contract(SampleABI)
> thesample=sample.new(10,{from:primary,data:SampleHEX,gas:3000000})
> samplerecpt=eth.getTransactionReceipt(thesample.transactionHash)
> samplecontract=sample.at(samplerecpt.contractAddress)
> samplecontract.get.call()
> samplecontract.set.sendTransaction(9, {from:primary, gas:3000000})
> samplecontract.get.call()
```

在图 8-24 和图 8-25 中, 我们设置之前通过 genesis 给定账户初始余额的账户作为主账户, 确认余额是 11 个 Ether。然后执行 loadScript 命令, 分别加载合约的 ABI 和二进制文件, 然后基于合约的 ABI, 并通过 eth.contract() 命令生成合约的接口定义。基于接口定义, 设置合约的初始值为 10, 通过设定创建者账户 from:primary、合约二进制代码, 创建合约所用的 Gas 上限, 完成合约的创建。

```
rodger@ubu-blockchain2:~/ethereum-docker/ethereum-docker/ethereum-testnet-docker/dockercomposefiles$ geth attach http://192.168.5.172:8201
Welcome to the Geth JavaScript console!

instance: Geth/01/v1.6.5-stable/linux-amd64/go1.8
coinbase: 0xc7f9a898d14f91fde9a3d37c150c6a4f9b104545
at block: 292 (Wed, 14 Jun 2017 08:04:03 PDT)
datadir: /ethcluster/779977/data/01
modules: admin:1.0 eth:1.0 net:1.0 rpc:1.0 web3:1.0

> primary=eth.accounts[0]
"0xc7f9a898d14f91fde9a3d37c150c6a4f9b104545"
> balance=web3.fromWei(eth.getBalance(primary),"ether")
11
> loadScript("./ethcontract/Sample.abi")
true
> loadScript("./ethcontract/Sample.bin")
true
> sample=eth.contract(SampleABI)
{
  abi: [
    {
      constant: true,
      inputs: [],
      name: "value",
      outputs: [{...}],
      payable: false,
      type: "function"
    },
    {
      constant: false,
      inputs: [{...}],
      name: "set",
      outputs: [],
      payable: false,
      type: "function"
    },
    {
      constant: true,
      inputs: [],
      name: "get",
      outputs: [{...}],
      payable: false,
      type: "function"
    },
    {
      inputs: [{...}],
      payable: false,
      type: "constructor"
    }
  ],
  eth: {
    accounts: ["0xc7f9a898d14f91fde9a3d37c150c6a4f9b104545"],
    blockNumber: 296,
    coinbase: "0xc7f9a898d14f91fde9a3d37c150c6a4f9b104545",
    compile: {
      ll: function(),
      serpent: function(),
      solidity: function()
    },
    defaultAccount: undefined,
    defaultBlock: "latest",
    gasPrice: 18000000000,
    hashrate: 0,
    mining: false,
    pendingTransactions: [],
  }
}
```

图 8-24 通过 cluster0 peer01 命令行执行合约部署和交易调用 (1)

```

pendingTransactions: [],
protocolVersion: "0x3f",
syncing: false,
call: function(),
contract: function(abi),
estimateGas: function(),
filter: function(fil, callback),
getAccounts: function(callback),
getBalance: function(),
getBlock: function(),
getBlockNumber: function(callback),
getBlockTransactionCount: function(),
getBlockUncleCount: function(),
getCode: function(),
getCoinbase: function(callback),
getCompilers: function(),
getGasPrice: function(callback),
getHashrate: function(callback),
getMining: function(callback),
getPendingTransactions: function(callback),
getProtocolVersion: function(callback),
getRawTransaction: function(),
getRawTransactionFromBlock: function(),
getStorageAt: function(),
getSyncing: function(callback),
getTransaction: function(),
getTransactionCount: function(),
getTransactionFromBlock: function(),
getTransactionReceipt: function(),
getUncle: function(),
getWork: function(),
iban: function(iban),
icapNamereg: function(),
isSyncing: function(callback),
namereg: function(),
resend: function(),
sendIBANTransaction: function(),
sendRawTransaction: function(),
sendTransaction: function(),
sign: function(),
signTransaction: function(),
submitTransaction: function(),
submitWork: function()
},
at: function(address, callback),
getData: function(),
new: function()
}
thesample=sample.new(10,{from:primary,data:SampleHEX,gas:3000000})
abi: [{
  constant: true,
  inputs: [],
  name: "value",
  outputs: [{...}],
  payable: false,
  type: "function"
}, {
  constant: false,

```

图 8-25 通过 cluster0 peer01 命令行执行合约部署和交易调用 (2)

创建合约命令执行后，会产生一个交易哈希值。通过这个交易哈希值，可以跟踪创建合约交易在区块链上的执行结果。图 8-25 通过命令查询这个合约创建交易哈希值对应的收据。如图 8-26 所示，交易在没有被打包进区块时，返回的结果为 null，一旦打包进区块，会得到一个 JSON 格式的返回值，其中包括了合约的地址信息。一旦一个合约部署到区块链上，通过 at 到这个合约地址，我们就可以获得链上这个合约的执行实例接口，基于合约接口就可以进行合约外部接口的调用，如图 8-27 所示。get 方法上 call() 只是读取本地节点的账本状态值，不会花费任何 Gas，而 set 接口上的 sendTransaction() 会将交易发送到网络上进行共识，需要花费 Gas，共识打包进区块后就可以完成所有网络对应状态值的修改。我们执行了把当前状态值从 10 修改为 9 的交易，在交易没有被共识打包进区块前，通过 get 获得的值依然是 10，直到交易被打包进区块，当前状态的值也随之被修改成了 9。

为了演示在其他节点上的合约调用，我们再打开一个终端。打开 cluster1 的 peer02 节点的控制台，直接 at 到上一个终端部署的智能合约地址并进行 set 操作，将之前的值 9 改成 6。







```

> primary=eth.accounts[0]
> loadScript("./ethcontract/Sample.abi")
> loadScript("./ethcontract/Sample.bin")
> sample=eth.contract(SampleABI)
> samplecontract=sample.at("0x8087fa310b15579f7e1423aafc199c10c91a5dfb")
> samplecontract.get.call()
> samplecontract.set.sendTransaction(6, {from:primary, gas:3000000})
> samplecontract.get.call()

```

图 8-28 至图 8-30 所示为执行过程。

```

rodger@ubu-blockchain2:~/ethereum-docker/ethereum-docker/ethereum-testnet-docker/dockercomposefiles$ geth attach http://192.168.5.172:9202
Welcome to the Geth JavaScript console!

instance: Geth/02/v1.6.5-stable/linux-amd64/go1.8
coinbase: 0x9ef6bb56f8d1f6899975776978a3c8f5d6753341
at block: 488 (Wed, 14 Jun 2017 08:25:29 PDT)
datadir: /ethcluster779977/data/02
modules: admin:1.0 eth:1.0 net:1.0 rpc:1.0 web3:1.0

> primary=eth.accounts[0]
"0x9ef6bb56f8d1f6899975776978a3c8f5d6753341"
> loadScript("./ethcontract/Sample.abi")
true
> loadScript("./ethcontract/Sample.bin")
true
> sample=eth.contract(SampleABI)
{
  abi: [
    {
      constant: true,
      inputs: [],
      name: "value",
      outputs: [{}],
      payable: false,
      type: "function"
    },
    {
      constant: false,
      inputs: [{}],
      name: "set",
      outputs: [],
      payable: false,
      type: "function"
    },
    {
      constant: true,
      inputs: [],
      name: "get",
      outputs: [{}],
      payable: false,
      type: "function"
    },
    {
      inputs: [{}],
      payable: false,
      type: "constructor"
    }
  ],
  eth: {
    accounts: ["0x9ef6bb56f8d1f6899975776978a3c8f5d6753341", "0x511a41c8a14260c1118040460157a48ede70eea4", "0xd25e70b2945b06b44f590f9b69865"],
    blockNumber: 507,
    coinbase: "0x9ef6bb56f8d1f6899975776978a3c8f5d6753341",
    compile: {
      lll: function(),
      serpent: function(),
      solidity: function()
    },
    defaultAccount: undefined,
    defaultBlock: "latest",
    gasPrice: 18000000000,

```

图 8-28 通过 cluster1 peer02 执行合约交易调用 (1)

再回到 cluster0 的 peer01 节点的控制台，通过 get.call() 查询到最新的状态值为 6。

值得注意的是，如果 docker compose 停止，若下次再想从上次区块继续进行下去，需要修改 docker compose 文件 geth\_cluster0 和 geth\_cluster1 的 INIT\_CONFIG 值为 N，然后再启动 docker compose，否则会重新产生创世区块，重新开始。



```

    hashrate: 75/3,
    mining: true,
    pendingTransactions: [],
    protocolVersion: "0x3f",
    syncing: false,
    call: function(),
    contract: function(abi),
    estimateGas: function(),
    filter: function(fil, callback),
    getAccounts: function(callback),
    getBalance: function(),
    getBlock: function(),
    getBlockNumber: function(callback),
    getBlockTransactionCount: function(),
    getBlockUncleCount: function(),
    getCode: function(),
    getCoinbase: function(callback),
    getCompilers: function(),
    getGasPrice: function(callback),
    getHashrate: function(callback),
    getMining: function(callback),
    getPendingTransactions: function(callback),
    getProtocolVersion: function(callback),
    getRawTransaction: function(),
    getRawTransactionFromBlock: function(),
    getStorageAt: function(),
    getSyncing: function(callback),
    getTransaction: function(),
    getTransactionCount: function(),
    getTransactionFromBlock: function(),
    getTransactionReceipt: function(),
    getUncle: function(),
    getWork: function(),
    iban: function(iban),
    iscapNamereg: function(),
    isSyncing: function(callback),
    namereg: function(),
    resend: function(),
    sendIBANTransaction: function(),
    sendRawTransaction: function(),
    sendTransaction: function(),
    sign: function(),
    signTransaction: function(),
    submitTransaction: function(),
    submitWork: function()
  },
  at: function(address, callback),
  getData: function(),
  new: function()
}
samplecontract-sample.at("0x8087fa310b15579f7e1423aaafc199c10c91a5dfb")
abi: [{
  constant: true,
  inputs: [],
  name: "value",
  outputs: [{...}],
  payable: false,
  type: "function"
}]

```

图 8-29 通过 cluster1 peer02 执行合约交易调用 (2)

```

    type: "function"
  }, {
    constant: false,
    inputs: [{...}],
    name: "set",
    outputs: [],
    payable: false,
    type: "function"
  }, {
    constant: true,
    inputs: [],
    name: "get",
    outputs: [{...}],
    payable: false,
    type: "function"
  }, {
    inputs: [{...}],
    payable: false,
    type: "constructor"
  }
],
address: "0x8087fa310b15579f7e1423aaafc199c10c91a5dfb",
transactionHash: null,
allEvents: function(),
get: function(),
set: function(),
value: function()
}
samplecontract.get.call()
> samplecontract.set.sendTransaction(6, {from: primary, gas: 3000000})
"0xb6bb7b2976bb4234c25e5bfff0e816676bd4ba52390f47f3bcb80f89e7f1827"
> samplecontract.get.call()
> samplecontract.get.call()
> samplecontract.get.call()
> samplecontract.get.call()
> samplecontract.get.call()
> samplecontract.get.call()
>

```

图 8-30 通过 cluster1 peer02 执行合约交易调用 (3)



## 8.4 以太坊适用场景剖析

以太坊作为公有链，可以在全球范围内的任何联网节点间形成共识，账本为所有节点共享访问。可以将以太坊理解为一个全球共享的开放全账本，任何人任何时候都可以访问任何账户地址上发生的任何一笔交易，任何人也可以查看任何合约地址上的合约内容，只要是需要共识的业务就可以。正是因为全球范围内采用工作量证明作为共识算法抵抗恶意节点，为了激励网络不断出块，也为了抵抗恶意 51% 攻击和 DoS 攻击，设计了在以太坊上通兑的以太币 Ether，维护网络正常运行的节点可以得到铸币和手续费奖励。只要网络整体算力足够大，想通过算力进行 51% 攻击的恶意节点都会得不偿失，以太坊执行智能合约交易的每一个操作码都需要支付 Gas 进而消耗以太币，操作码按照对资源消耗不同征收不同的燃料费用，实施 DoS 攻击也会带来很大的经济成本，这就保障了以太坊网络的整体安全性和稳定性。但也正是这些安全机制设计的制约，使得执行每一个交易都涉及 Gas 和 Ether 的计算，乃至整个状态 Patricia 树根哈希的计算又采用 PoW 进行资源消耗型哈希运算，从整体上降低了出块和验证的速度。

### 1. 以太坊不适用的场景

#### (1) 不适用于高吞吐率要求的业务

以太坊从 2015 年 7 月 30 日正式运行，到 2017 年 7 月 26 日总交易数为 40 701 017（注：数据来源<sup>①</sup>），平均每天 55 908 笔交易，平均每小时 2330 笔，0.65 笔/秒，正常情况下 7~15 笔/秒。需要上千笔 TPS 的业务目前显然是不适合以太坊的。

#### (2) 不适用于响应时间敏感的业务

目前以太坊平均出块时间 14 秒，按照社区定义的标准“家园”（Homestead）以太坊稳妥起见需要 120 个块进行确认，则交易确认的时间需要差不多 30 分钟，遇到网络拥塞，交易甚至几个小时都会处于 pending 状态，因此对于响应时间要求高的业务是不适合的。

#### (3) 不适用于需要权限访问控制的业务

目前以太坊是全开放的，任何人都可以访问和操作网络，甚至破坏网络。要限定用户的操作权限，只有在部署的合约内进行控制，比如，拥有特定账户私钥的用户可以执行特定的交易。

#### (4) 不适用于对交易隐私性要求高的业务

目前以太坊的所有合约和交易都是公开的，任何人都可以查到任何地址上发生的任何交易，合约的内容也是公开的。虽然交易者身份同交易地址之间是没有必然联系的，但是黑客也可以结合社会工程实施攻击获得交易者的身份。对于要求隐匿交易内容的商业应用显然不适合。

#### (5) 暂不适合经济价值巨大的业务

由于以太坊还处于起步阶段，又是公开透明全开放账本的网络，合约内容、交易内容

<sup>①</sup> 参考 <https://etherscan.io/chart/tx>。



可以被任何人看到并做漏洞分析，如果某个合约存在巨大的经济价值，则可能会被攻击而导致极大的经济损失。如 TheDAO 被攻击，其众筹总额达 1.5 亿美金之巨，造成了巨大的影响。采用智能合约的项目要本着循序渐进、不断完善探索的态度去推进。

## 2. 以太坊应用分类

一般来讲，在以太坊上可以实现各种公开的、面向大众的业务逻辑，可以有三大类应用<sup>①</sup>。

第一类是金融应用，为用户提供更强大的用他们的钱管理和参与合约的方法，包括子货币（代币）、金融衍生品、对冲合约、储蓄钱包、遗嘱，甚至一些种类全面的雇佣合约。

第二类是半金融应用，这里有钱的存在但也有很重的非金钱的方面，一个完美的例子是为解决计算问题而设的自我强制悬赏。

最后，还有在线投票和去中心化治理这样的非金融应用。然而，受限于上面提到的不适合的非功能性要求高的场景，如对于那些有高吞吐、低延时、隐私要求高的业务，建议采用许可链方案实现。

## 8.5 小结

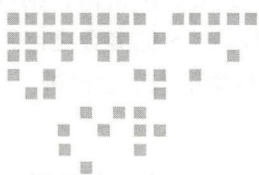
本章介绍了以太坊的出现背景、关键概念，详细说明了以太坊的核心架构。另外回顾了以太坊整个分叉历史和至今发生的大事件，对以太坊智能合约和流行的高级合约语言 Solidity 也做了说明，并且提供了一个以太坊测试网络构建，并在网络上编译、部署和运行智能合约的案例，让读者有更加深刻的实战感性认识和经验。最后结合以太坊的特点，说明了整个平台生态的适用性。

### 习题

- [1] 以太坊区别于比特币的最大特点是什么？
- [2] 以太坊有哪两种账户？它们有什么不同？
- [3] 以太坊为什么采用账户模型？账户模型是如何组织，如何保证不可篡改的？
- [4] 以太坊的区块结构是什么？最多可以包括几个叔区块头？为什么要包括叔区块？
- [5] 以太坊采用了一种改进型 Merkle 树用来存储和组织账户状态列表、交易列表等？这种树叫什么？这种树增加了哪些节点类型，访问效率如何？

---

① Vitalik Buterin. Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform. 2013a. URL{  
<http://ethfans.org/posts/ethereum-whitepaper>}.



## Chapter 9

## 第 9 章

## 超级账本 Fabric

作者：张海宁 陈家豪

以比特币为代表的加密数字货币获得了巨大成功，人们也渐渐意识到区块链技术的潜在价值，它不仅可以用作比特币的底层技术，还能够应用到更多的业务场景中。但是对于商业应用来说，公有链尚存在一些技术问题，例如比特币有如下局限性：

- 交易效率低，整个网络吞吐量大约只有每秒 7 笔左右；
- 交易确认时间长，大约需要 60 分钟才能确认；
- 交易没有最终性（finality），从理论上讲，每个区块都没有最终确定；
- 交易没有隐私性，任何人都可以在链上查看到每笔交易。

比特币等公有链的不足之处制约了其商业应用的场景，设计适合商用的区块链平台迫在眉睫。在各界强烈的呼声中，Linux 基金会于 2015 年 12 月启动了名为“超级账本”（Hyperledger）的开源项目，旨在推动各方协作，共同打造基于区块链的企业级分布式账本底层技术，用于构建支撑业务的行业应用和平台。超级账本项目提供多种区块链技术框架和代码，包含开放的协议和标准、不同的共识算法和存储模型，以及身份认证、访问控制和智能合约等服务。模块化、性能和可靠性是很重要的设计目标，以便支持各式各样的商业应用场景。

从创始成员看，参与超级账本项目的公司阵容相当豪华，不仅有 IBM、Intel 等科技巨头，也有摩根大通、富国银行等金融大鳄，还有 R3、ConsenSys 等专注于区块链的公司。截至 2018 年 1 月，超级账本项目已经汇集了全球超过 200 家公司，不管是从代码数量还是从社区参与度来看，超级账本都是最大的区块链开源项目，是由大企业领衔的商业化联盟链项目。

超级账本里包括若干个不同的项目（project），每个项目根据发展程度可处于 5 个阶



段，分别是：提案、孵化、活跃、弃用和终止。截至2018年6月，通过提案进入孵化或活跃阶段的项目共有10个。区块链框架类项目有5个：Fabric、Sawtooth、Iroha、Burrow 和 Indy；区块链工具类项目有5个：Cello、Composer、Explorer、Quilt 和 Caliper。

超级账本联盟成立之后，IBM 把原先“开放区块链”（Open Blockchain）项目的代码贡献给了 Linux 基金会，成为 Fabric 代码的主要组成部分。Fabric 于2017年7月发布了1.0 GA 版本，正式进入活跃阶段，并得到社区较为广泛的使用。

本章主要介绍超级账本中 Fabric 项目架构原理和开发应用的方法，涵盖 Fabric 的总体架构、架构解析、Fabric 应用的方法等内容。

## 9.1 Fabric 基础架构

Fabric（编织品）项目的目标是实现一个通用的许可链的底层基础框架。为了适用于不同的场合，Fabric 采用模块化架构提供可切换和可扩展的组件，包括共识算法、加密安全、数字资产、记录仓库、智能合约和身份鉴权等服务。Fabric 克服了比特币等公有链项目的缺陷，如吞吐量低、交易公开无隐私性、无最终确定性以及共识算法低效等问题，使得用户能够方便地开发商业应用。但是，Fabric 作为新的权限链的项目，也存在不足之处，如1.0的共识算法尚不支持BFT类型、在交易过程还有并发控制的局限性、整体性能还有待提高等。这些问题在下文中会进行讨论。

### 9.1.1 架构概述

Fabric 的总体架构分为网络层、核心层、服务层以及接口层，如图9-1所示。



图 9-1 Fabric 总体架构图

网络层由多个分布式节点组成。这些节点构成了一个点对点（P2P）的网络，采用 Gossip 协议进行节点间互相发现和数据传输，并采用 gPRC 的框架互相调用接口功能。

核心层中的共识机制是区块链系统的核心模块，它确保各个节点对数据达成共识。Fabric 1.0 仅支持用于开发测试的 SOLO 模式和用于生产系统的 Kafka 方式。其中 Kafka 实现的是 CFT（Crash Fault Tolerant）的容错类型，需要假定联盟链网络中没有故意作恶的节

点。Fabric 还允许用插拔的方式增加 BFT (Byzantine Fault Tolerant) 的容错类型。

区块链的存储主要包含以文件形式存储的链式区块数据, 以及在数据库保存的键值对 (Key-Value Pair) 状态数据。其中链式区块数据存放的是交易的原始数据区块, 通过区块的哈希值形成防篡改的链式结构。状态数据库的作用主要是加速对数据的访问。因为区块链数据采用链式顺序存放, 在读取数据时通常需要遍历整个链的数据块, 采用数据库能够从索引迅速定位到所需数据。

Fabric 中的智能合约称为“链码” (chain code)。链码部署在节点上, 采用容器技术形成隔离的运行环境。链码的生命周期管理主要包括链码的安装、实例化、调用和终止等。

作为联盟链方案, Fabric 包含管理成员身份的功能。参与区块链网络的成员身份必须是明确的, 成员之间知道彼此组织身份信息, 每个交易都有确定的参与方和背书方, 这是绝大多数商用系统的需求。相比之下, 许多公有链的用户身份是匿名的, 参与方无须确认身份信息。

服务层利用核心层的基础功能, 封装成服务的形式, 提供给应用端来使用。账本服务和交易服务通过核心层的共识算法和区块链存储, 实现基本的区块链数据操作能力。链码服务提供智能合约的功能封装。事件服务则提供应用侦听系统事件并处理的功能。权限服务根据成员和用户的身份信息, 对其操作权限进行控制, 成为商业应用中安全管理的一部分。

接口层的目的是使 Fabric 的应用 (客户端) 能够方便地调用区块链的服务。接口主要以 API 的形式提供, 能够完成通道、链码、交易等方面的操作。为了便于编程语言的调用, Fabric 提供了绑定不同语言的 SDK, 如 Node 和 Java 等的 SDK。此外, Fabric 还提供了命令行接口 CLI, 可无须编程, 通过命令直接调用 Fabric 的功能。

### 9.1.2 主要组件

Fabric 的组件包括客户端 (Client)、网络节点 (Peer)、CA (Certificate Authority) 节点和排序节点 (Orderer)。

客户端的主要作用是和 Fabric 系统交互, 实现对区块链系统的操作。这些操作分为管理类和链码类。管理类操作包括启停节点和配置网络等; 链码类操作主要是链码的生命周期管理, 如安装、实例化以及调用链码。最常用的客户端是命令行客户端 (CLI), 此外是用 Fabric SDK 开发的应用客户端。用户可以通过不同的客户端使用 Fabric 系统的功能。

网络节点 (Peer) 是区块链去中心化网络中的对等节点, 按照功能主要分为背书节点 (Endorser) 和确认节点 (Committer)。背书节点主要对交易预案进行校验、模拟执行和背书。确认节点主要负责检验交易的合法性, 并更新和维护区块链数据与账本状态。在实际部署中, 背书节点和确认节点既可以部署在同一物理节点上, 也可以分开部署。

排序节点 (Orderer) 的主要职责是对各个节点发来的交易进行排序。在并发的情况下, 各个节点交易的先后时序需要通过排序节点来确定并达成共识。排序节点按照一定规则确



定交易顺序之后,发给各个节点把交易持久化到区块链的账本中。排序节点支持互相隔离的多个通道,使得交易只发送给相关的节点(Peer)。

CA 节点主要给 Fabric 网络中的成员提供基于数字证书的身份信息,可以生成或取消成员的身份证书(certificate)。在成员身份明确的基础上,Fabric 可以实现权限控制的管理。

Fabric 网络的组件往往归属于不同的组织,在组织之间形成对等的去中心化网络。每个组织通常拥有自己的客户端、网络节点和 CA 节点,并且可以根据需要创建一个或多个不同的类型节点。排序节点不属于某个组织的实体,属于组织共同维护的组件。各个组件的相互关系如图 9-2 所示。

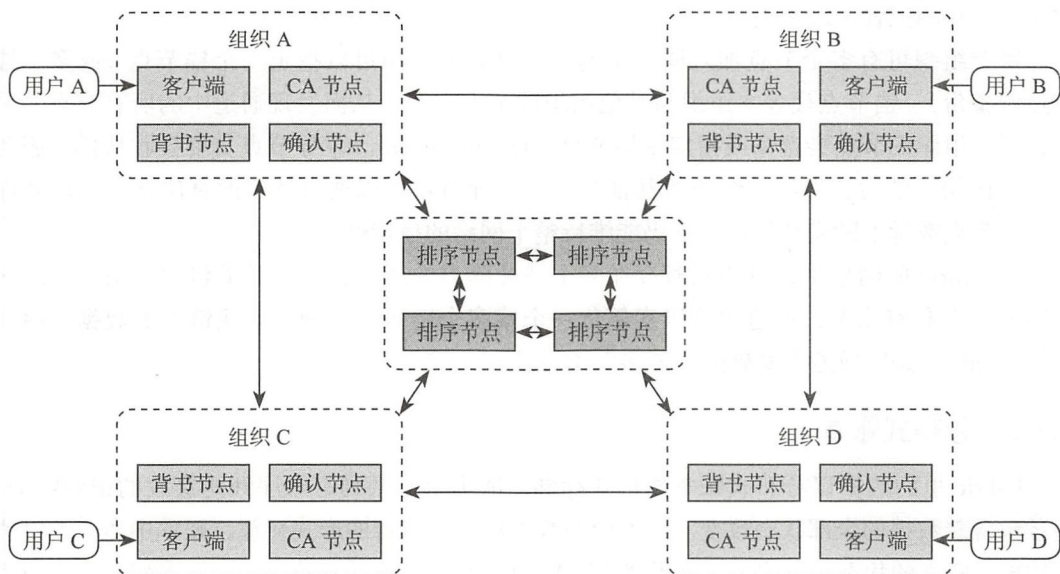


图 9-2 Fabric 组件关系

### 9.1.3 P2P 网络

Fabric 的节点组成了一个 P2P 的网络。P2P 网络的节点可以通过直接交换来共享资源和服务。节点彼此之间处于对等的地位,并不依赖于集中的服务节点来进行资源调度。网络中的每一个节点既能充当网络服务的请求者,又能响应其他节点的请求,提供网络资源和服务,以达到资源共享的目的。

在 Fabric 中,P2P 技术主要用于网络节点之间的健康检测和账本同步。节点间的 P2P 网络由 Gossip 协议实现,在通道中的各个节点会持续广播和接收 Gossip 消息。Gossip 的内容包含节点的状态、账本数据以及通道数据等信息,由于 Gossip 消息需要发送者的签名,因此伪造的消息很容易被识别。此外,还可以基于消息的签名对信息进行隔离。通过 Gossip 协议,受延迟、网络分区或其他因素而导致账本没有同步的节点,最终都会同步到最新的账本状态。

### 9.1.4 通道

商业应用的一个重要需求是私密性交易，为此 Fabric 设计了通道（Channel）来提供成员之间的隐私保护。通道是部分网络成员之间拥有独立的通信渠道，在通道中发送的交易只有属于通道的成员才可见，因此通道可以看作是 Fabric 的网络中部分成员的私有通信“子网”。

通道由排序服务管理。在创建通道的时候，需要定义它的成员、组织、锚节点（anchor peer）和排序服务的节点。一条和通道对应的区块链结构也同时生成，用于记录账本的交易。通道的初始配置信息记录在区块链的创世块（第一个区块）中。通道的配置信息可以用增加一个新的配置区块来更改。

每个组织可有多个节点加入同一个通道。这些节点中可以指定一个锚节点（或多个锚节点做备份）。锚节点代表本组织与其他组织的节点交互，从而发现通道中的所有节点。另外，同一组织的节点会选举或指定主导节点（leading peer）。主导节点负责接收从排序服务发来的区块，然后转发给本组织的其他节点。主导节点可以通过特定的算法选出，因此保证了在节点数量不断变动的情况下仍能维持整个网络的稳定性。

在 Fabric 的网络中，可能同时存在多个彼此隔离的通道，每个通道包含一条私有的区块链和一个私有账本，通道中可以实例化一个或多个链码，以操作区块链上的数据。由此可见，Fabric 是以通道为基础的多链多账本系统。

### 9.1.5 分布式账本

Fabric 里的数据以分布式账本的形式存储。账本由一系列有顺序和防篡改的记录组成，记录包含着数据的全部状态改变。账本中的数据项以键值对的形式存放，账本中所有的键值对构成了账本的状态，也称为“世界状态”（World State）。每个通道中有唯一的账本，由通道中所有成员共同维护。每个确认节点上都保存了它所属通道的账本的一个副本，因而是分布式账本。对账本的访问需要通过链码实现对账本键值对的增加、删除、更新和查询等操作。

账本由区块链和状态数据库 2 部分组成。

区块链是一组不可更改的、有序的区块（数据块），记录着全部交易的日志。每个区块中包含若干个交易的数据，不同区块包含的交易数量可以不同。区块之间用哈希链（Hashed-link）关联：每个区块头包含该区块所有交易的哈希值，以及上一个区块头的哈希值。这样的链式架构可以确保每个区块的数据不可更改，以及每个区块之间的顺序关系不可更改。这个特点决定了区块链的区块只可以添加在链的尾部。

状态数据库记录了账本中所有键值对的当前值，相当于对当前账本的交易日志做了索引。链码执行交易的时候需要读取账本的当前状态，从状态数据库可以迅速获取键值的最新状态。如果没有状态数据库，要获得某个键值时，需要遍历整个区块链中和该键值相关的交易，效率非常低。因此，读取状态数据库可以认为是快速定位和访问某个键值的方法。另外，当状态数据库出现故障的时候，可以通过遍历账本重新生成。状态数据库如图 9-3 所示。



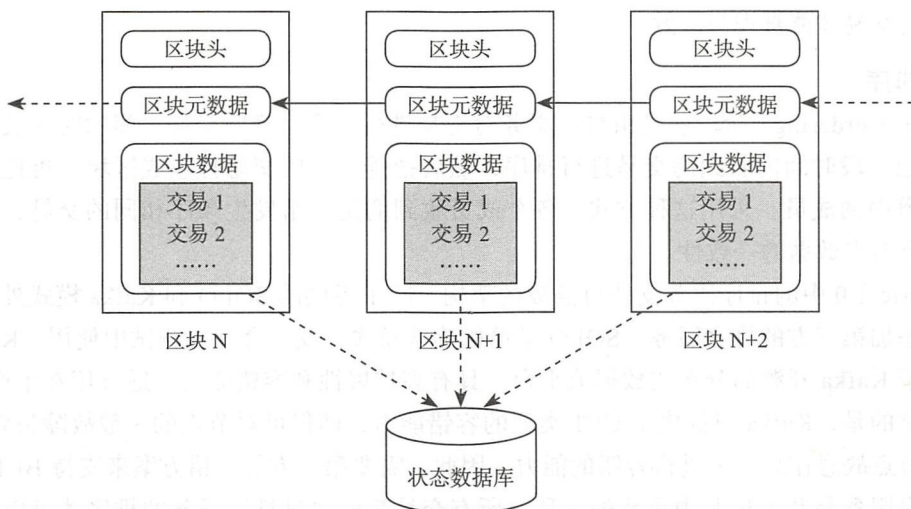


图 9-3 状态数据库

当一个区块附加到区块链尾部的时候，如果区块中的有效交易修改了键值对，则会在状态数据库中做相应的更新，这样区块链和状态数据库就能始终保持一致。

区块链的数据块以文件形式保存在各个节点中。状态数据库原理上可以是各种键值数据库，Fabric 缺省使用的是 LevelDB，它也支持 CouchDB 的选项。CouchDB 除了支持键值数据之外，也支持 JSON 格式的文档模型，能够做复杂的查询。

### 9.1.6 共识机制

Fabric 的网络节点本质上是互相复制的状态机，节点之间需要保持相同的账本状态。为了实现这个目的，各个节点需要通过共识（consensus）过程，对账本状态的变化达成一致性的认同。因为账本记录的是系统中发生的交易，共识机制实际上就是要确保各个节点看到相同的交易顺序和交易内容，从而保证各个节点都处于相同的状态。

Fabric 的共识过程包括 3 个阶段：背书、排序和校验。

#### 1. 背书

在背书（endorsement）阶段，背书节点对客户端发来的交易预案进行合法性检验，然后模拟执行链码得到交易结果，最后根据设定的背书逻辑判断是否支持该交易预案。如果背书逻辑决定支持交易预案，它将把预案签名后发回给客户端。缺省情况下，背书节点的背书逻辑是直接支持预案并签名。但是节点也可以依照业务规则设定背书逻辑，从而仅对符合业务需求的交易进行背书。如果背书节点判定不支持交易，则给客户端返回错误信息。

客户端通常需要根据链码的背书策略，向一个或者多个成员的背书节点发出背书请求。背书策略会定义需要哪些节点背书交易才有效，例如需要 5 个成员的背书节点中至少 3 个同意，或者某个特殊身份的成员支持等。客户端只有在收集满足背书策略的支持之后，广



播出去的交易才能被视为有效。

## 2. 排序

排序 (ordering) 阶段就是由排序服务对交易进行排序, 确定交易之间的时序关系。排序服务把一段时间内收到的交易进行排序, 然后把排序后的交易打包成区块, 再把区块广播给通道中的成员。采用这种方式, 各个成员收到的是一组发生顺序相同的交易, 从而保证了所有节点数据的一致性。

Fabric 1.0 中的排序服务支持可插拔的架构。除了提供的 SOLO 和 Kafka 模式外, 用户还可以添加第三方的排序服务。SOLO 是单机确认模式, 仅适合开发测试中使用。Kafka 模式是基于 Kafka 开源的分布式数据流平台, 具有高扩展性和容错能力, 适合用在生产系统。需要注意的是, Kafka 只提供了 CFT 类型的容错能力, 即仅可对节点的一般故障失效容错, 缺乏对节点故意作恶行为进行容错的能力。因此, 需要第三方的容错方案来支持 BFT。

排序服务是共识机制中重要的一环, 所有交易都要通过排序服务的排序才可以达成全网共识, 因此排序服务要避免成为网络上的性能瓶颈。为此, 排序服务采用轻量级设计, 只完成确定交易顺序的功能, 不参与其他操作。

## 3. 校验

校验 (validation) 阶段是确认节点对排序后的交易进行一系列的检验, 包括交易数据的完整性检查、是否重复交易、背书签名是否符合背书策略的要求、交易的读写集是否符合多版本并发控制 MVCC (Multiversion Concurrency Control) 的校验等。当交易通过了所有校验之后, 将被标注为合法并写入账本中。因为所有的确认节点都按照相同的顺序检验交易, 并且把合法的交易依次写入账本中, 所以它们的状态能够始终保持一致。

更多的共识机制的介绍参见 9.1.9 节。

### 9.1.7 智能合约 (链码)

“智能合约”的概念最早出现在 1996 年, 由密码学家尼克·萨博 (Nick Szabo) 首次提出。他对智能合约的定义为: “智能合约是一套以数字形式定义的承诺 (promises), 包括合约参与方可以在上面执行这些承诺的协议。”

智能合约能够部署和运行在区块链环境中, 由一段代码来描述相关的业务逻辑。部署后的智能合约在区块链中无法修改。智能合约的执行完全由代码决定, 不受人为因素的干扰。一般来说, 参与方通过智能合约规定各自的权利和义务、触发合约的条件以及结果, 一旦该智能合约在区块链环境中运行就可以得出客观、准确的结果。

在 Fabric 中, 智能合约也称链码 (chaincode), 分为用户链码和系统链码 2 种, 通常所说的链码指的是用户链码。链码是访问账本的基本方法, 一般是用 Go 等高级语言编写的、实现规定接口的代码。上层应用可以通过调用链码来初始化和管理工作账本的状态。只要有适当的权限, 链码之间也可以互相调用。





链码安装在背书节点上，需要在某个通道上实例化并且定义相应背书策略后才能运行。链码部署后不可更改，但是可以通过升级来发布新的功能或修复问题。在 Fabric 的设计中，链码运行在一个安全的 Docker 容器沙盒内。该容器由背书节点创建和管理，以便隔离背书节点和链码的运行环境。

### 9.1.8 成员服务提供者

Fabric 与其他公有链系统的重要区别在于，Fabric 具有成员身份和权限管理的能力。成员服务提供者 MSP (Membership Service Provider) 抽象地描述了参与者在网络中的身份，成员服务是成员服务提供者的具体实现。参与者在网络中的身份由参与者的成员服务描述，成员服务最重要的一个部件是参与者所持有的身份证书。该证书由特定 CA 签发，当若干参与者的身份证书可追溯到同一个根 CA 时，则认为这些参与者处于同一个信任链中，这些参与者也称该信任链的成员 (member)，这些成员可用组织 (organization) 来代表。组织中的成员可以分为普通成员和管理员 2 种角色，管理员角色拥有对组织配置进行修改的权限。

在实际运行中，参与者通过成员服务可对其他参与者进行身份认证、授权访问权限等操作以管理区块链网络，从而对账本数据的访问控制可以在更广泛的网络和通道层面上进行操控，有助于实现数据的授权访问和隐私保护。例如：可以允许特定参与者调用链码应用程序，但阻止他们部署新的链码；制定规则以拒绝验证由某些参与者签名的交易等。

### 9.1.9 交易流程

Fabric 正常的交易流程如图 9-4 所示，步骤如下。

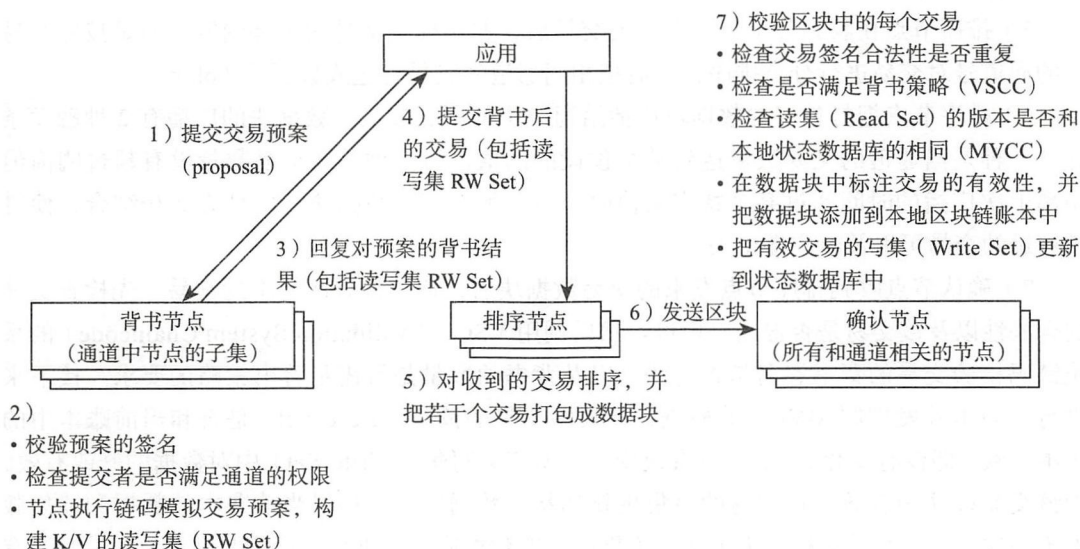


图 9-4 交易流程



1) 应用端首先构建交易的预案, 预案的作用是调用通道中的链码来读取或者写入账本的数据。应用端使用 Fabric 的 SDK 打包交易预案, 并使用用户的私钥对预案进行签名。应用打包完交易预案后, 接着把预案提交给通道中的背书节点。通道的背书策略定义了哪些节点背书后交易才能有效。应用端根据背书策略选择相应的背书节点, 并向它们提交交易预案。

2) 背书节点收到交易预案后, 首先校验交易的签名是否合法, 然后根据签名者的身份, 确认其是否具有权限进行相关交易。此外, 背书节点还需要检查交易预案的格式是否正确以及之前是否提交过(防止重放攻击)。在所有合法性校验通过后, 背书节点按照交易预案调用链码。链码执行时, 读取的数据(键值对)是节点中本地的状态数据库。需要指出的是, 链码在背书节点中是模拟执行, 即对数据库的写操作并不会对账本造成改变, 所有的写操作将归总到一个写入的集合(Write Set)中记录下来。如果在链码中对某个数据项写完之后再读取, 得到的数据是在链码执行前的旧值, 因为新值并没有实际写入账本中。在链码执行完成后, 将返回链码读取过的数据集(Read Set)和链码写入的数据集(Write Set)。读集和写集将在确认节点中用于确定交易是否最终写入账本。

3) 背书节点把链码模拟执行后得到的读写集(Read-Write Set)等信息签名后发回给预案提交方(应用端)。

4) 应用端在收到背书响应之后, 检查背书节点的签名和比较不同节点背书的结果是否一致。如果预案是查询账本的请求, 则应用端无须提交交易给排序节点。如果是更新账本请求, 应用端在收集到满足背书策略的背书响应数量之后(例如, 背书策略规定的至少5个背书节点中的4个背书响应), 把背书预案中得到的读写集、所有背书节点的签名和通道号发给排序节点。

5) 排序节点在收到各个节点发来的交易后, 并不检查交易的全部内容, 而是按照交易中的通道号对交易进行分类排序, 然后把相同通道的交易打包成数据块(blob)。

6) 排序节点把打包好的数据块广播给通道中的所有成员。数据块的广播有2种触发条件: 一种是当通道的交易数量达到某个预设的阈值; 另一种是在交易数量没有超过阈值但距离上次广播的时间超过某个特定阈值时, 也可触发广播数据块。两种方式相结合, 使得排序过的交易可以及时广播出去。

7) 确认节点收到排序节点发来的交易数据块后, 逐笔检查区块中的交易。先检查交易的合法性以及该交易是否曾经出现过。然后调用 VSCC (Validation System Chaincode) 的系统链码检验交易的背书签名是否合法, 以及背书的数量是否满足背书策略的要求。接下来进行多版本并发控制 MVCC 的检查, 即校验交易的读集(Read Set)是否和当前账本中的版本一致(即没有变化)。如果没有改变, 说明交易写集(Write Set)中对数据的修改有效, 把该交易标注为有效, 将交易的写集更新到状态数据库中。如果当前账本的数据和读集版本不一致, 则该交易被标注为无效, 不更新状态数据库。数据块中的交易数据在标注成“有效”或“无效”后封装成区块(block)写入账本的区块链中。





上述的交易流程中，Fabric 分离了背书节点和确认节点的职责，增加了系统的吞吐量，使系统更具可扩展性。在交易的确认过程中，采用了 MVCC 的乐观锁（optimistic locking）模型，提高了系统的并发能力。需要注意的是，MVCC 也带来了一些局限性。例如，在同一个区块中若有两个交易先后对某个数据项做更新，顺序在后的交易将失败，因为它的读集版本和当前数据项版本已经不一致了（因为之前的交易更新了数据）。

## 9.2 架构详细原理

本节将详细介绍 Fabric 的主要技术特点，包括成员身份及权限管理、通道原理和链码开发方法。

### 9.2.1 成员身份管理

Fabric 属于许可链，与公有链最大的区别在于要求参与者先注册身份，该身份即为参与者在区块链网络中的标识。在 Fabric 中，身份证书是每个参与者在区块链网络中的标识，同时也是权限管理的基础，证书采用了 X.509 标准，并且通过椭圆曲线密码学算法来生成公私钥对。

#### 1. 成员服务提供者组件

成员服务提供者组件 MSP 是对身份证书的抽象表达。网络中每个参与者都拥有 MSP，它包含了参与者的身份证书、数字签名、验证算法以及若干判断身份是否有效的规则。基于 MSP，参与者可以向网络中的其他节点发送或者验证签名消息。

在 Fabric 中，通道是一个非常重要的概念。通道实现了对数据的隔离，只有授权的网络实体才能访问通道内的数据。通常每个通道会定义一个或多个 MSP 实体用于控制数据的访问权限，其中每个 MSP 都对应着一个根 CA 或者中间 CA，只有由 MSP 对应的 CA 签发的身份证书才能通过相应 MSP 的身份校验。

#### 2. 节点 MSP

节点 MSP 定义了节点在网络中的身份，相关内容如下。

##### （1）节点的身份属性

在 Fabric 中，每个成员的身份都有一些特殊的属性以进行某些权限操作，如某个身份的 attrs 中有 hf.Registrar.Roles 字段且该字段的值为“client, user, peer”，则说明该身份可以签发其他身份证书，可签发的证书类型为 client、user 和 peer。除了上述的 hf.Registrar.Roles 外，还有 hf.Revoker 和 hf.IntermediateCA 两个常用属性分别对应该实体能否注销用户和签发中间 CA。

##### （2）节点 MSP 基本组成

在 Fabric 启动 peer 节点或排序节点前，需要先配置 localMspId 环境变量来指定节点的



MSP 名称。其次是 mspConfigPath 环境变量，该变量为节点设定了 MSP 存放的目录，节点运行时会从该目录加载相应的 MSP 数据。

MSP 的目录结构包括如下子目录（以下提及的证书均采用 X.509 标准）。

- ❑ cacerts：存放一组 CA 的自签名根证书，这些证书构成整个组织证书信任的基础。
- ❑ intermediatecerts（可选）：存放一组受信任的中间证书，这组证书充当了中间 CA，可用于验证证书的合法性，由根证书签发。
- ❑ admincerts：存放一组 MSP 管理员的身份证书，管理员拥有权限修改本 MSP 的配置，如删除 cacerts 目录中的某些证书。
- ❑ signcerts：存放本节点的身份证书，身份证书必须通过本组织的 CA 签发。
- ❑ keystore：存放与本节点身份证书对应的私钥，用于对消息签名。
- ❑ tlscacerts：存放用于 TLS 通信协议的自签名证书。
- ❑ crls（可选）：存放若干已经撤销的身份证书。
- ❑ config.yaml（可选）：存放组织单元列表。若身份证书中的 ou 域包含在 ous 中，且由 cacerts 中的根证书签发，则该身份证书有效。

假设存在 MSP 实体 Org1MSP，若某个网络实体要通过 Org1MSP 校验，则该实体的证书必须满足以下条件：

- ❑ 证书链的根在 Org1MSP 的 cacerts 目录中；
- ❑ 证书不在 Org1MSP 的 crls 目录中；
- ❑ 若 Org1MSP 定义了组织单元，则证书中的 ou 域必须包含组织单元中定义的一个或多个元素。

### （3）生成节点的 msp 目录

如果需要生成节点的上述 msp 目录文件，可使用 cryptogen 工具和 Fabric CA server 2 种方法。下面分别介绍这 2 种方法的使用步骤。

#### 1) 使用 cryptogen 工具生成 msp 目录。

cryptogen 是 Fabric 自带的一个命令行工具，可根据文件 crypto-config.yaml 来快速生成若干数量的成员证书。一个简单的 crypto-config.yaml 例子如下：

```
OrdererOrgs:
- Name: Orderer
  Domain: orgorderer1
  Template:
    Count: 1
PeerOrgs:
- Name: Org1
  Domain: org1
  Template:
    Count: 1
- Name: Org2
  Domain: org2
```





```
Template:
Count: 1
```

其中 OrdererOrgs 和 PeerOrgs 关键字用于区分组织 (organization) 的类型。两种组织的内部结构如下。

- OrdererOrgs 中定义 orderer 组织的信息, Name 字段定义了组织的名称, Domain 字段定义了组织的域名。同时, 通过 template 字段可批量生成 orderer 节点。上述例子中定义了一个名字为 Orderer、域名为 orgorderer1 的组织, 并且它指定 template 中 count 的数值为 1, 则在该组织下只有一个 orderer, 其 ID 为 orderer0。
- PeerOrgs 中定义 peer 组织的信息。上述例子定义了两个组织, 分别为 Org1 和 Org2, 对应的域名为 org1、org2。与 orderer 类似, 每个组织生成了一个 peer, 虽然 org1 中 peer0 和 org2 中 peer0 的 ID 重复, 但是它们不属于同一组织, 因此通过域名很容易就能区分出来。

更多配置信息请读者自行参考源码<sup>①</sup>中关于 cryptogen 的描述。

通过上面定义的 crypto-config.yaml 文件, 运行 cryptogen, 会生成 crypto-config 目录。该目录的结构如下:

```
crypto-config
|--- ordererOrganizations
|       |--- orgorderer1
|               |--- msp
|               |--- ca
|               |--- tlsca
|               |--- users
|               |--- orderers
|               |--- orderer0.orgorderer1
|                       |--- msp
|                       |--- tls
|--- peerOrganizations
|       |--- org1
|               |--- msp
|               |--- ca
|               |--- tlsca
|               |--- users
|               |--- peers
|               |--- peer0.org1
|                       |--- msp
|                       |--- tls
|       |--- org2
|               |--- msp
|               |--- ca
|               |--- tlsca
```

① <http://github.com/hyperledger/fabric/common/tools/cryptogen/main.go>.



```

|--- users
|--- peers
|--- peer0.org2
|   |--- msp
|   |--- tls

```

可以看出，每个组织都包含了 msp、ca、tlsca 和 users 目录，然后根据组织类型的不同还分别有 peers 和 orderers 目录，里面存放着组织中每个成员的 msp 和 tls 文件。

在编辑好 crypto-config.yaml 文件之后，可用如下命令生成 msp 目录：

```
$ cryptogen generate --config=./crypto-config.yaml
```

## 2) 通过 Fabric-CA sever 生成 msp 目录。

如 9.1.2 节中所介绍的，CA 节点是每个组织给本组织成员提供数字证书的身份信息的服务。CA 节点中运行的主要是 Fabric-CA server 这个服务。该服务启动的方法有 2 种，分别是通过 fabric-ca-server 二进制可执行文件直接启动和通过 Docker 容器镜像启动。无论选择哪种方式，都可以很方便地配置 server 的参数。通常与 fabric-ca-server 配套使用的还有客户端 fabric-ca-client，该客户端程序可以方便用户与 CA server 之间的交互。下面介绍 2 种启动 Fabric-CA Server 的方法。

### ① 使用 fabric-ca-server 可执行文件启动 CA 服务。

下面以 Ubuntu16.0.4 为例。请先确保 Go 语言已经安装并且设置 GOPATH 环境变量。

#### ❑ 安装 libtool 和 libtdhl-dev:

```
$ sudo apt install libtool libtdhl-dev
```

#### ❑ 安装 Fabric-CA server 和 Fabric-CA client:

```
$ go get -u github.com/hyperledger/fabric-ca/cmd/...
```

#### ❑ 创建 test\_ca 文件夹，存放 Fabric-CA server 的配置文件:

```
$ mkdir $GOPATH/src/test_ca && cd $GOPATH/src/test_ca
```

#### ❑ 初始化 Fabric-CA server:

```
$ fabric-ca-server init -b admin:adminpw
```

该命令在当前目录创建 CA server 的证书、私钥和数据库等文件，其中 fabric-ca-server-config.yaml 为 CA server 配置文件。配置文件提供了丰富的参数设置以满足不同的需求。

参数 -b 为 server 设定管理员的初始 ID 和 secret。管理员可从客户端通过 admin:adminpw 来获取默认管理员的根证书，以进行后续的证书签发、证书撤销等操作。

#### ❑ 启动 Fabric-CA server:

```
$ fabric-ca-server start
```

该命令读取当前目录下 fabric-ca-server-config.yaml 配置文件，载入相应的证书和私钥





并启动 CA server 服务。

### ② 通过 Docker 容器方式启动 CA 服务。

先确保系统已安装 17.03 或以上版本的 Docker 程序，以及 1.11 或以上版本的 docker-compose 程序。

❑ 从 Docker Hub 上下载 fabric-ca 镜像：

```
$ docker pull hyperledger/fabric-ca:x86_64-1.0.0
```

❑ 创建 test\_ca 文件夹，存放 Fabric-CA server 的配置文件：

```
$ mkdir $GOPATH/src/test_ca && cd $GOPATH/src/test_ca
```

❑ 创建 docker-compose.yaml 文件：

```
fabric-ca-server:
  image: hyperledger/fabric-ca:x86_64-1.0.0
  container_name: fabric-ca-server
  ports:
    - "7054:7054"
  environment:
    - FABRIC_CA_HOME=/etc/hyperledger/fabric-ca-server
  volumes:
    - "./fabric-ca-server:/etc/hyperledger/fabric-ca-server"
  command: bash -c 'while true; do sleep 3; done'
```

❑ 启动 CA 服务：

```
$ docker-compose -f docker-compose.yaml up -d
$ docker exec fabric-ca-server fabric-ca-server start -b admin:adminpw
```

当容器启动后，会在当前目录下新建 fabric-ca-server 目录以存放 CA Server 的证书、私钥配置文件等。

### ③ fabric-ca-client 客户端的使用方法。

使用 fabric-ca-client 命令可以方便地跟 Fabric CA Server 进行交互。一些常用的操作如下。

❑ 登录管理员账户，获取证书签发权限。

上述启动 CA Server 的流程中定义了管理员的 ID 和密码分别为 admin 和 adminpw，通过以下命令登录管理员用户：

```
$ export FABRIC_CA_CLIENT_HOME=$HOME/fabric-ca/clients/admin
$ fabric-ca-client enroll -u http://admin:adminpw@localhost:7054
```

执行上面的命令，会在 \$FABRIC\_CA\_CLIENT\_HOME 目录下生成一个 msp 目录和一个 fabric-ca-client-config.yaml 配置文件。

❑ 注册成员 peer1：

```
$ export FABRIC_CA_CLIENT_HOME=$HOME/fabric-ca/clients/admin
```



```
$ fabric-ca-client register --id.name peer1 --id.type peer --id.secret peer1pw
```

在上述命令中, 通过 id.name、id.type、id.secret 分别声明了注册用户的名称、类型和登录密码。

❑ 登录新注册的 peer1:

```
$ export FABRIC_CA_CLIENT_HOME=$HOME/fabric-ca/clients/peer1
$ fabric-ca-client enroll -u http://peer1:peer1pw@localhost:7054
```

上述的第一条命令通过环境变量 FABRIC\_CA\_CLIENT\_HOME 为 peer1 指定了主目录 \$HOME/fabric-ca/clients/peer1, 该目录用于存放 Fabric server 返回的文件。第二条命令用于获取 peer1 用户的证书文件, 该命令会在 \$HOME/fabric-ca/clients/peer1 目录下生成一个 msp 目录, 其中 msp 目录存放着 peer1 的证书和对应的私钥。

❑ 注销成员 peer1:

```
$ export FABRIC_CA_CLIENT_HOME=$HOME/fabric-ca/clients/admin
$ fabric-ca-client revoke -e peer1
```

上述命令使用 admin 的身份来注销成员, 参数 -e 指定需要被注销的成员。

关于 fabric-ca-server 配置参数的详细描述请参考链接<sup>①</sup>; 关于 fabric-ca-client 配置的参数的详细描述请参考链接<sup>②</sup>。

#### (4) 通道 MSP

除了节点需要 MSP 之外, 通道也需要 MSP。在通道创建时, 需要先定义通道有哪些组织参与。在通道中的每个组织都由相应的 MSP 表示, 同时每个 MSP 都有各自的信任基础, 即 CA, 包括根 CA 和中间 CA。只有由 MSP 对应的 CA 签发的身份证书才能通过相应 MSP 的身份校验。在通道中定义 MSP 可以实现对通道中数据的访问权限控制。

### 9.2.2 通道的结构

通道是 Fabric 中非常重要的概念, 它实质上是由排序节点划分和管理的私有原子广播通道, 目的是对通道的信息进行隔离, 使得通道外的实体无法访问通道内的信息, 从而实现交易的隐私性。

目前通道分为系统通道 (System Channel) 和应用通道 (Application Channel) 2 种。排序节点通过系统通道来管理应用通道, 用户的交易信息通过应用通道传递。对一般用户来说, 通道是指应用通道。系统通道与应用通道的关系如图 9-5 所示。

① <http://hyperledger-fabric-ca.readthedocs.io/en/latest/serverconfig.html>.

② <http://hyperledger-fabric-ca.readthedocs.io/en/latest/clientconfig.html>.



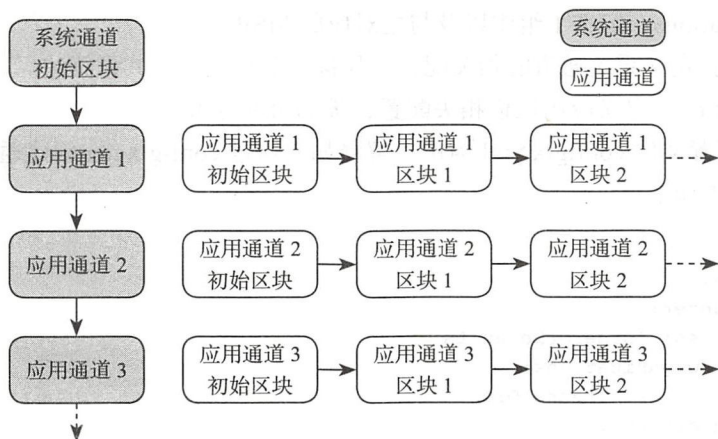


图 9-5 系统通道与应用通道

通道由排序服务节点负责管理，同时该节点还负责排序通道中的交易。在通道中一般包含有若干成员（组织）。若两个网络实体的身份证书能够追溯到同一个根 CA，则认为这两个实体属于同一组织。此外，通道中的每个组织都会有一个或以上的“锚节点”，它负责与其他组织交换共享账本的数据。

创建通道的时候定义了成员，只有通过成员 MSP 验证的实体，才能够加入到通道并访问通道数据。一个验证例子如下。

Org1 是通道 mychannel 的成员之一，与 Org1 绑定的 MSP 标识为 Org1MSP，其代表的 CA 称为 CA1。若实体的 MSP 满足以下条件则认为实体有权限访问 mychannel 的数据：

- 实体的 MSP 标识（ID）为 Org1MSP；
- 实体身份证书的信任链源头为 CA1。

实体只要满足通道中任意成员的 MSP 校验，就认为该实体有权限访问通道中的数据。

### 1. 通道的配置

通道的配置信息都被打包到一个区块中，并存放在通道的共享账本中。该区块除了配置信息外不包含其他交易信息，称为通道的配置区块（Configuration Block）。通道可以使用配置区块来更新配置，因此在账本中每新添加一个配置区块，通道就按照最新配置区块的定义来修改配置。通道账本的首个区块一定是配置区块，也称为创世区块（Genesis Block）。

### 2. 使用 configtxgen 工具生成通道的配置

configtxgen 是 Fabric 提供的工具，用于生成通道所需要的配置文件。configtxgen 工具以一个 yaml 文件作为输入，一般称为 configtx.yaml。该文件定义了将要创建通道的配置信息，该文件通常包括以下部分。

1) Profiles：包含了通道的配置模板，通过 configtxgen 工具的参数 -profile 来指定使用哪个模板。

2) Organizations: 定义了组织以及与之对应的 MSP。

3) Orderer: 定义系统通道的相关配置, 如排序节点地址、共识算法。

4) Application: 定义应用通道相关配置, 被 profile 引用。

以下面的配置文件 configtx.yml 为例, 解释如何通过 configtxgen 创建通道的初始区块。  
configtx.yml 清单如下:

```
Profiles:
  Genesis:
    Orderer:
      <<: *OrdererDefaults
      Organizations:
        - *OrdererOrg
    Consortiums:
      SampleConsortium:
        Organizations:
          - *PeerOrg1
          - *PeerOrg2
  Channel:
    Consortium: SampleConsortium
    Application:
      <<: *ApplicationDefaults
      Organizations:
        - *PeerOrg1
        - *PeerOrg2
Organizations:
  - &OrdererOrg
    Name: OrdererOrg
    ID: OrdererMSP
    MSPDir: OrdererOrg/msp
  - &PeerOrg1
    Name: PeerOrg1MSP
    ID: PeerOrg1MSP
    MSPDir: PeerOrg1/msp
    AnchorPeers:
      - Host: peer0.peerOrg1.com
        Port: 7051
  - &PeerOrg2
    Name: PeerOrg2MSP
    ID: PeerOrg2MSP
    MSPDir: PeerOrg2/msp
    AnchorPeers:
      - Host: peer0.peerOrg2.com
        Port: 7051
Orderer: &OrdererDefaults
OrdererType: solo
Addresses:
```



```

- orderer.ordererOrg.com:7050      #orderer 地址
BatchTimeout: 2s
BatchSize:
  MaxMessageCount: 10
  AbsoluteMaxBytes: 98 MB
  PreferredMaxBytes: 512 KB
Kafka:
Organizations:

Application: &ApplicationDefaults
  Organizations:

```

Profiles 字段定义如下:

```

Profiles:
  Genesis:
    Orderer:
      <<: *OrdererDefaults
      Organizations:
        - *OrdererOrg
    Consortiums:
      SampleConsortium:
        Organizations:
          - *PeerOrg1
          - *PeerOrg2
  Channel:
    Consortium: SampleConsortium
    Application:
      <<: *ApplicationDefaults
      Organizations:
        - *PeerOrg1
        - *PeerOrg2

```

上面的 profile 定义了系统通道和应用通道 2 种不同类型的通道。

系统通道必须定义 Orderer 和 Consortiums 2 部分。

❑ Genesis.Orderer.Organizations 字段定义参与此系统通道的组织信息, 另外 Genesis.Orderer 还引用了文件后面 Orderer 字段定义的内容 <<: \*OrdererDefaults, 通过该引用定义系统通道的配置信息, 如共识算法、orderer 地址等。

❑ Genesis.Consortiums 字段定义 Orderer 为哪些联盟提供服务, 可以理解为只有联盟中的组织能够创建通道。

应用通道必须定义 Application 和 Consortium 2 部分。

❑ Channel.Application 主要定义了应用通道内的组织信息。

❑ Channel.Consortium 指定了与应用通道相关联联盟的名称。

Organizations 字段定义如下:

```

Organizations:

```

```

- &OrdererOrg
  Name: OrdererOrg
  ID: OrdererMSP
  MSPDir: OrdererOrg/msp

- &PeerOrg1
  Name: PeerOrg1MSP
  ID: PeerOrg1MSP
  MSPDir: PeerOrg1/msp
  AnchorPeers:
    - Host: peer0.peerOrg1.com
      Port: 7051

- &PeerOrg2
  Name: PeerOrg2MSP
  ID: PeerOrg2MSP
  MSPDir: PeerOrg2/msp
  AnchorPeers:
    - Host: peer0.peerOrg2.com
      Port: 7051

```

Organizations 字段定义了 2 种类型的组织，分别是 Orderer 类型和普通类型。Orderer 类型的组织包括 MSP 名称、MSP 的 ID 和相应的 msp 目录；普通类型组织的定义与 Orderer 类型相似，但每个普通类型组织都要定义一个锚节点，用于代表本组织与其他组织通信。

Orderer 字段定义如下：

```

Orderer: &OrdererDefaults
  OrdererType: solo
  Addresses:
    - orderer.ordererOrg.com:7050      #orderer 地址
  BatchTimeout: 2s
  BatchSize:
    MaxMessageCount: 10
    AbsoluteMaxBytes: 98 MB
    PreferredMaxBytes: 512 KB
  Kafka:
  Organizations:

```

上面定义了通道中排序节点使用的共识算法（solo）、排序节点地址、创建批量交易的超时时间（2 秒）、每个区块最多交易数（10）、区块的大小限制（98MB）、区块大小建议（512KB）。

Application 字段定义如下：

```

Application: &ApplicationDefaults
  Organizations:

```

以上 Application 没有定义组织信息，但是 profile 在引用该字段时可以自由添加组织信息。



定义好 yaml 文件后, 需要把 configtxgen 工具以及 msp 目录都复制到 yaml 文件所在的目录下, configtxgen 默认会读取当前目录的 configtx.yaml 作为输入。

#### 1) 创建排序节点的初始区块:

```
configtxgen -profile Genesis -outputBlock genesis.block
```

该命令通过 profile 参数来指定生成 yaml 文件中 Profile.Genesis 的配置, 通过 -outputBlock 参数来将区块写入 genesis.block 文件。

#### 2) 创建应用通道 mychannel 的初始区块的交易文件 channel.tx:

```
configtxgen -profile Channel -outputCreateChannelTx channel.tx -channelID mychannel
```

该命令通过 -outputCreateChannelTx 参数将生成的交易写入 channel.tx 文件, 通过 -channelID 来指定创建通道的名称为 mychannel。

#### 3) 创建配置区块的交易文件 Org1MSPanchors.tx, 以更新 mychannel 中 PeerOrg1 的锚节点:

```
configtxgen -profile Channel -outputAnchorPeersUpdate Org1MSPanchors.tx -channelID mychannel -asOrg PeerOrg1MSP
```

该命令通过 -asOrg 来指定使用 PeerOrg1MSP 身份创建配置区块, 并且通过 -outputAnchorPeersUpdate 参数将配置区块写入到文件 Org1MSPanchors.tx 中。

类似地, 创建配置区块的交易文件 Org2MSPanchors.tx 以更新 mychannel 中 PeerOrg2 的锚节点:

```
configtxgen -profile Channel -outputAnchorPeersUpdate Org2MSPanchors.tx -channelID mychannel -asOrg PeerOrg2MSP
```

### 3. 通道相关命令

对通道的管理可通过命令行的方式。与通道相关的命令如下。

- ❑ peer channel create : 用于创建通道, 主要参数有 -c、-f、-o, 分别用于指定通道 ID、configtx 的路径和 orderer 的地址。
- ❑ peer channel fetch : 抓取通道中的特定区块, 通过 -c 和 -f 参数来指定通道 ID 和 orderer 地址。
- ❑ peer channel join : 加入通道, 通过 -b 参数指定初始区块。
- ❑ peer channel list : 列出 peer 加入的通道。
- ❑ peer channel update : 签名并且发送 configtx 以升级通道配置, 需要通过 -c、-f、-o 参数分别指定通道 ID、configtx 的路径以及排序节点的地址。

### 4. 动态修改通道配置

在通道创建后, 通道相关的配置以区块的形式存在于通道的账本中。如果需要修改通

道的配置，可通过生成新的配置区块去更新。修改通道配置的步骤如下：

- ❑ 通过 SDK 或 CLI 获得最新的配置区块；
- ❑ 编辑配置区块；
- ❑ 计算配置更新量；
- ❑ 为配置区块添加配置更新量；
- ❑ SDK 或 CLI 签名并发送配置区块。

若新的配置区块通过验证，则通道配置以最新配置区块为准。

由于获取的配置区块以二进制的 proto 格式返回，导致配置区块的信息难以直接阅读。一般通过 Fabric 自带的 configtxlator 工具来把配置区块转化为 JSON 格式用以阅读和编辑。当编辑完成后再次使用工具把修改好的 JSON 文件转化为 proto 格式，最后把 proto 格式的配置区块发回给排序节点验证。具体操作流程请参考 9.3.11 节修改通道配置的内容。

### 9.2.3 链码

Fabric 的链码是一个重要的概念，它是由 Go 等语言编写并且满足特定接口的程序，具有图灵完备性。链码根据交易内容对账本状态进行管理。一般链码运行的业务逻辑由网络中的成员一起协商。链码运行在背书节点上的 Docker 容器沙盒中，以便与背书节点的其他进程隔离。

#### 1. 链码的生命周期管理

链码完整的生命周期如图 9-6 所示，包括初始状态、等待安装、等待实例化、运行和停止等几种状态。

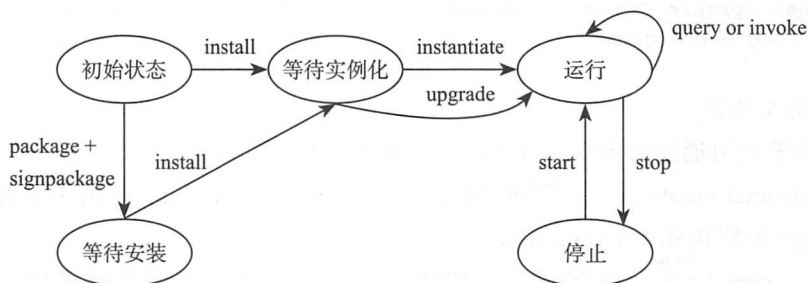


图 9-6 链码生命周期图

Fabric 提供命令行工具管理链码的生命周期，常用命令如下。

- ❑ `peer chaincode install`：把链码打包成可部署格式，并将其存入到背书节点的文件系统目录下（`CORE_PEER_FILESYSTEMPATH/chaincode`）。
- ❑ `peer chaincode instantiate`：把安装到背书节点上的链码实例化到指定的通道上。该命令会在节点上创建运行链码的 Docker 容器，并初始化链码。
- ❑ `peer chaincode invoke`：调用指定链码，若执行交易的节点还没创建运行链码的容器，



则背书节点会先创建该容器再执行交易。

- ❑ `peer chaincode query` : 查询指定链码, 若执行交易的节点还没创建运行链码的容器, 则背书节点会先创建该容器再执行交易。该交易只查询节点上的状态, 不生成区块。
- ❑ `peer chaincode package`: 把链码打包成可部署格式。
- ❑ `peer chaincode signpackage`: 签名打包后的链码。
- ❑ `peer chaincode upgrade` : 升级链码, 需要先用 `peer chaincode install` 命令安装最新的代码, 然后使用本命令来升级已经实例化的代码。

## 2. 链码的背书策略

链码实例化时可指定背书策略。当确认节点接收到交易时, 节点获知相关链码信息, 然后检查该链码的背书策略, 判断交易是否满足背书策略, 若满足则标注交易为合法。

### (1) 背书策略的设计

背书策略可分为主体 `principal` (`P`) 和阈值 `threshold` (`T`) 2 部分, 具体如下。

1) `principal` 指定由哪些成员进行背书。

2) `threshold` 接受两个输入, 分别为阈值  $t$  和若干个  $P$  的集合  $n$ , 只要交易中包含了  $n$  中  $t$  个成员的背书则认为交易合法。

例如:

❑ `T (1, 'A', 'B')` 则需要 A、B 中任意成员背书。

❑ `T (1, 'A', T (2, 'B', 'C'))` 则需要 A 成员背书或 B、C 成员同时背书。

### (2) 背书策略相关的 CLI 命令

`peer` 命令中使用布尔表达式来描述背书策略。`principal` 中的成员用 `MSP.ROLE` 的方式描述, 其中 `MSP` 为成员的 `MSPID`, `ROLE` 则为成员的角色, 分为 `member` (成员) 和 `admin` (管理员)。如 `peerOrg1.admin` 则要求 `peerOrg1` 组织中的管理员进行背书; `peerOrg1.member` 只需要 `peerOrg1` 中任一成员进行背书即可。

在 `peer` 命令中设置背书策略的语法如下:

```
EXPR(E[, E...])
```

其中 `EXPR` 是布尔表达式 `AND` 或 `OR`, 而 `E` 要么是一个 `principle`, 要么是另一个 `EXPR`。

例如:

❑ `AND ('Org1.member', 'Org2.member')` 要求 `Org1` 和 `Org2` 的成员同时背书。

❑ `OR ('Org1.admin', AND ('Org2.member', 'Org3.member'))` 要求 `Org1` 的管理员背书或 `Org2`、`Org3` 的成员同时背书。

## 3. 链码开发

链码开发过程中需要实现链码接口。交易的类型决定了哪个接口函数将会被调用, 如 `instantiate` 和 `upgrade` 类型会调用链码的 `Init` 接口, 而 `invoke` 类型的交易则调用链码的

Invoke 接口。链码的接口定义如下：

```
type Chaincode interface {
    Init(stub ChaincodeStubInterface) pb.Response
    Invoke(stub ChaincodeStubInterface) pb.Response
}
```

下面通过一个例子讲解链码的开发流程。示例链码根据交易的类型创建键值对并记录到账本中，或者根据键名到账本中查找与之相对应的值。

请先确保 Go 语言环境已经安装并且正确设置 GOPATH 环境变量。

### (1) 创建链码存放目录

创建 keyValueStore 目录以存放链码，同时进入目录：

```
mkdir $GOPATH/src/keyValueStore
cd $GOPATH/src/keyValueStore
```

创建并编辑链码文件 keyValueStore.go。完整代码如下：

```
package main

import (
    "fmt"
    "github.com/hyperledger/fabric/core/chaincode/shim"
    "github.com/hyperledger/fabric/protos/peer"
)

type KeyValueStore struct {
}

func (t * KeyValueStore) Init(stub shim.ChaincodeStubInterface) peer.Response {
    args := stub.GetStringArgs()
    if len(args) != 2 {
        return shim.Error("Incorrect arguments. Expecting a key and a value")
    }
    err := stub.PutState(args[0], []byte(args[1]))
    if err != nil {
        return shim.Error(fmt.Sprintf("Failed to create asset: %s", args[0]))
    }
    return shim.Success(nil)
}

func (t *KeyValueStore) Invoke(stub shim.ChaincodeStubInterface) peer.Response {
    fn, args := stub.GetFunctionAndParameters()

    var result string
    var err error
    if fn == "set" {
        result, err = set(stub, args)
    } else {
```



```

        result, err = get(stub, args)
    }
    if err != nil {
        return shim.Error(err.Error())
    }
    return shim.Success([]byte(result))
}

func set(stub shim.ChaincodeStubInterface, args []string) (string, error) {
    if len(args) != 2 {
        return "", fmt.Errorf("Incorrect arguments. Expecting a key and a value")
    }
    err := stub.PutState(args[0], []byte(args[1]))
    if err != nil {
        return "", fmt.Errorf("Failed to set asset: %s", args[0])
    }
    return args[1], nil
}

func get(stub shim.ChaincodeStubInterface, args []string) (string, error) {
    if len(args) != 1 {
        return "", fmt.Errorf("Incorrect arguments. Expecting a key")
    }
    value, err := stub.GetState(args[0])
    if err != nil {
        return "", fmt.Errorf("Failed to get asset: %s with error: %s", args[0], err)
    }
    if value == nil {
        return "", fmt.Errorf("Asset not found: %s", args[0])
    }
    return string(value), nil
}

func main() {
    if err := shim.Start(new(KeyValueStore)); err != nil {
        fmt.Printf("Error starting KeyValueStore chaincode: %s", err)
    }
}

```

## (2) 链码源代码分析

### 1) 导入头文件。

链码必须依赖 chaincode shim 包和 peer protobuf 包，它们分别用于链码的控制与数据传输。其次定义 KeyValueStore 类型，作为 chaincode shim 的载体。

```
package main
```



```
import (
    "fmt"
    "github.com/hyperledger/fabric/core/chaincode/shim"
    "github.com/hyperledger/fabric/protos/peer"
)
```

```
type KeyValueStore struct {
}
```

## 2) 实现 Init 方法。

Init 方法通过 shim.ChaincodeStubInterface 接口来获取实例化链码交易的相关信息。该接口的 GetStringArgs 方法可获取交易传给链码的参数。链码实例化时接收 key 和 value 两个参数，因此先对参数个数进行验证。若验证通过，则将第一个和第二个参数分别作为 key 和 value 存入到账本中。

把状态存入账本需要借助 shim.ChaincodeStubInterface 接口 PutState 方法来完成。由于账本中的数据都以键值对的形式储存，因此该方法也只接受 key、value 两个参数。其中 value 为 byte 格式，里面还包含多个 JSON 格式的键值对。

由于执行结果需要以消息的形式返回给客户端，因此还需要把返回消息封装成 fabric/protos/peer 中的 Response 格式。

值得注意的是，链码升级的时候都会调用 Init 方法，编写升级链码时应注意 Init 方法的实现，以避免重新初始化或覆盖上一版本的账本状态。

```
func (t * KeyValueStore) Init(stub shim.ChaincodeStubInterface) peer.Response {
    args := stub.GetStringArgs()
    if len(args) != 2 {
        return shim.Error("Incorrect arguments. Expecting a key and a value")
    }
    err := stub.PutState(args[0], []byte(args[1]))
    if err != nil {
        return shim.Error(fmt.Sprintf("Failed to create asset: %s", args[0]))
    }
    return shim.Success(nil)
}
```

## 3) 实现 Invoke 方法。

与 Init 方法类似，Invoke 方法通过 shim.ChaincodeStubInterface 的 GetFunctionAndParameters 方法来获取 invoke 交易的参数。其中返回的 fn 与 args 分别为交易调用的具体函数名以及相应参数。此时 Invoke 方法进一步判断 fn 的值以进行下一步操作 (set 或者 get)，并把操作结果存放在 result 变量中以返回操作结果。

```
func (t *KeyValueStore) Invoke(stub shim.ChaincodeStubInterface) peer.Response {

    fn, args := stub.GetFunctionAndParameters()

    var result string
```





```

var err error
if fn == "set" {
    result, err = set(stub, args)
} else {
    result, err = get(stub, args)
}
if err != nil {
    return shim.Error(err.Error())
}
return shim.Success([]byte(result))
}

```

为了完成对账本的读写，链码还需要实现以下 2 个方法。

□ set: 把输入的键值对记录在账本中。

□ get: 根据键读取账本中与之对应的值。

4) 实现 get 和 put 方法。

正如前面所说，invoke 方法根据 fn 的值来执行相应的 get 或 put 函数。这两个函数也需要 shim.ChaincodeStubInterface 接口来访问账本数据。

```

func set(stub shim.ChaincodeStubInterface, args []string) (string, error) {
    if len(args) != 2 {
        return "", fmt.Errorf("Incorrect arguments. Expecting a key and a value")
    }

    err := stub.PutState(args[0], []byte(args[1]))
    if err != nil {
        return "", fmt.Errorf("Failed to set asset: %s", args[0])
    }
    return args[1], nil
}

```

```

func get(stub shim.ChaincodeStubInterface, args []string) (string, error) {
    if len(args) != 1 {
        return "", fmt.Errorf("Incorrect arguments. Expecting a key")
    }

    value, err := stub.GetState(args[0])
    if err != nil {
        return "", fmt.Errorf("Failed to get asset: %s with error: %s", args[0],
err)
    }
    if value == nil {
        return "", fmt.Errorf("Asset not found: %s", args[0])
    }
    return string(value), nil
}

```



## 5) 实现主函数 main()。

链码需要在 main 函数中调用 shim.Start ( ) 方法用于链码的部署。

```
func main() {
    if err := shim.Start(new(KeyValueStore)); err != nil {
        fmt.Printf("Error starting KeyValueStore chaincode: %s", err)
    }
}
```

## (3) 测试链码

链码的测试需要通过完整的 Fabric 网络。使用官方提供的例子可以快速构建测试网络，从而简化链码的开发流程。这里介绍搭建测试网络的步骤。

1) 参考 9.3.1 节安装示例代码库。

2) 进入 fabric-samples 目录：

```
$ cd $GOPATH/src/github.com/hyperledger/fabric-samples
```

3) 把新编写的链码放入 fabric-samples 的 chaincode 目录下：

```
$ cp -r $GOPATH/src/keyValueStore ./chaincode
```

4) 进入 chaincode-docker-devmode 目录并启动网络，命令中会创建一个名为 myc 的通道：

```
$ cd chaincode-docker-devmode
$ docker-compose -f docker-compose-simple.yaml up -d
```

5) 进入 chaincode 容器，编译并运行链码：

```
$ docker exec -it chaincode
$ cd keyValueStore && go build
$ export CORE_PEER_ADDRESS=peer:7051
$ export CORE_CHAINCODE_ID_NAME=mycc:0
$ ./keyValueStore
$ exit
```

6) 进入 CLI 容器并初始化链码，链码 ID 为 mycc，版本号为 0，部署的通道名称是 myc：

```
$ docker exec -it cli bash
$ peer chaincode install -p chaincodedev/chaincode/keyValueStore -n mycc -v 0
$ peer chaincode instantiate -n mycc -v 0 -c '{"Args":["a","10"]}' -C myc
```

7) Invoke 和 Query 链码：

```
$ peer chaincode query -n mycc -c '{"Args":["query","a"]}' -C myc
$ peer chaincode invoke -n mycc -c '{"Args":["set","a","20"]}' -C myc
$ peer chaincode query -n mycc -c '{"Args":["query","a"]}' -C myc
```

正常情况下，2 次 query 返回的结果分别为 10 和 20。





开发链码时可以通过上述过程进行测试，但需避免使用相同的链码 ID，以免链码实例化失败。另外，对于链码升级来说，链码的 ID 应该保持不变，同时新链码的版本号需要比先前实例化的版本高，并通过 upgrade 交易来更新链码在通道中的状态。

假设对链码 `keyValueStore.go` 进行了更改，并把最新的链码保存在 `$GOPATH/src/keyValueStoreNew` 下，则升级链码的操作如下。

1) 进入 `fabric-samples` 目录并拷贝最新链码到 `chaincode` 目录：

```
$ cd $GOPATH/src/fabric-samples
$ cp -r $GOPATH/src/keyValueStoreNew ./chaincode
```

2) 进入 `chaincode` 容器，编译并运行更新后的链码：

```
$ docker exec -it chaincode bash
$ cd keyValueStoreNew && go build
$ export CORE_PEER_ADDRESS=peer:7051
$ export CORE_CHAINCODE_ID_NAME=mycc:1
$ ./keyValueStoreNew
$ exit
```

3) 进入 `cli` 容器并升级链码：

```
$ docker exec -it cli bash
$ peer chaincode install -p chaincodedev/chaincode/keyValueStoreNew -n mycc -v 1
$ peer chaincode upgrade -n mycc -v 1 -c '{"Args":["a","10"]}' -C myc
```

至此升级链码完毕，可以对最新的链码 `mycc` 进行操作。

#### (4) ChaincodeStubInterface 常用的接口方法

- ❑ `GetTXID`：获取交易的 ID；
- ❑ `GetCreator`：获取交易发起者的信息；
- ❑ `GetState (key)`：获取账本中的 `key` 值；
- ❑ `PutState (key, value)`：往账本中的 `key` 写入 `value`；
- ❑ `DelState (key)`：删除账本中的 `key`。

方法详细描述请参考源码<sup>①</sup>。

## 9.3 应用开发流程

本节将借助官方例子介绍如何构建完整的 Fabric 网络，包括成员结构的定义、节点的启动以及通道的创建等。在网络搭建完毕后，将阐述如何在网络上部署一个车辆登记的应用，并开发与之相应的 Web 用户界面。

由于例子中的应用通道中只有一个组织参与，因此在本节将会探讨如何修改通道配置，

① 推荐阅读 <https://github.com/hyperledger/fabric/blob/release/core/chaincode/shim/interfaces.go>。



以达到动态添加组织的目的。

本节的例子使用 Fabric 1.0.0 版本。

### 9.3.1 前期准备

假设读者已准备运行环境为 Ubuntu 16.04 的主机，并正确安装 docker、docker-compose、golang 以及 git 等常用软件。

1) 从下面地址下载本文例子的代码库。例子源自 Fabric 官方示例代码。为了说明开发原理，笔者做了少量修改。

```
$ cd $GOPATH/src
$ git clone https://github.com/LordGoodman/fabric-samples.git
```

2) 下载 Fabric 常用工具集：

```
$ cd ~/Downloads
$ curl https://nexus.hyperledger.org/content/repositories/releases/org/hyperledger/
fabric/hyperledger-fabric/linux-amd64-1.0.0/hyperledger-fabric-linux-amd64-1.0.0.tar.gz
| tar xz
```

3) 把工具集加入到 PATH 环境变量中：

```
$ echo 'export PATH=$PATH:$HOME/Downloads/bin' >> ~/.profile
$ source ~/.profile
```

4) 下载 Fabric 的 Docker 镜像：

```
$ cd $GOPATH/src/fabric-samples/scripts && bash fabric-preload.sh
```

### 9.3.2 定义 Fabric 集群

为了部署车辆登记应用，下面将启动一个简单的 Fabric 集群。该集群仅包含了一个 peer 节点（同时具有背书节点和确认节点的功能）、一个排序节点以及一个 CA 组织。peer 节点的证书由该 CA 组织签发。

#### 1. 证书以及通道的初始区块生成

启动 Fabric 集群之前，先使用 cryptogen 和 configtxgen 来生成必要的身份证书（存放在 crypto-config 目录）、通道初始区块（存放在 config 目录）等文件，\$GOPATH/src/fabric-samples/basic-network 目录下的配置文件 crypto-config.yaml 定义了 9.3.2 节中描述的集群。同时，在同一目录下的 configtx.yaml 文件定义了只包含一个组织的应用通道，链码将会在该通道中部署。

默认情况下，\$GOPATH/src/fabric-samples/basic-network 下已经预先生成好了 crypto-config 和 config 目录。用户想要重新生成这两个目录可以运行 generate.sh 脚本。该脚本会先删除目录，然后再通过 cryptogen、configtxgen 以及相应的配置文件重新生成。





## 2. 编写 peer 的 docker-compose 文件

由于 Fabric 中的节点都通过 Docker 容器来运行，因此当文件准备好后，还需要编写 docker-compose.yaml 文件。该文件定义了容器所使用的镜像以及容器内的环境变量。在 \$GOPATH/src/fabric-samples/basic-network 目录下的 docker-compose.yaml 文件已经定义好了本章所需的集群，请读者自行参考。下面以该文件中 peer 容器配置为例进行简要分析。

```
peer0.org1.example.com:
  container_name: peer0.org1.example.com
  image: hyperledger/fabric-peer
  environment:
    - CORE_PEER_ID=peer0.org1.example.com
    - CORE_LOGGING_PEER=debug
    - CORE_CHAINCODE_LOGGING_LEVEL=DEBUG
    - CORE_PEER_LOCALMSPID=Org1MSP
    - CORE_PEER_MSPCONFIGPATH=/etc/hyperledger/msp/peer/
    - CORE_PEER_ADDRESS=peer0.org1.example.com:7051
  working_dir: /opt/gopath/src/github.com/hyperledger/fabric
  command: peer node start
  ports:
    - 7051:7051
    - 7053:7053
  volumes:
    - /var/run:/host/var/run/
    - ./crypto-config/peerOrganizations/org1.example.com/peers/peer0.org1.example.com/msp:/etc/hyperledger/msp/peer
    - ./config:/etc/hyperledger/configtx
  depends_on:
    - orderer.example.com
  networks:
    - basic
```

上述文件定义了名为 peer0.org1.example.com 的容器。容器使用的镜像为 hyperledger/fabric-peer:1.0.0，而 environment 字段则定义了容器中的环境变量。各环境变量的作用如表 9-1 所示（更多参数及其作用请参考源码<sup>①</sup>）。

表 9-1 environment 字段的变量

| 环境变量名称                       | 作用                              |
|------------------------------|---------------------------------|
| CORE_PEER_ID                 | 设置 Peer 节点在网络中的标识               |
| CORE_LOGGING_PEER            | 设置 Peer 节点日志等级                  |
| CORE_CHAINCODE_LOGGING_LEVEL | 设置与 Peer 节点相关的 chaincode 容器日志等级 |
| CORE_PEER_LOCALMSPID         | 设置 Peer 节点的 MSP 标识              |
| CORE_PEER_MSPCONFIGPATH      | 设置 Peer 节点的 MSP 目录              |
| CORE_PEER_ADDRESS            | 设置 Peer 节点的监听地址                 |

① <https://github.com/hyperledger/fabric/blob/release/sampleconfig/core.yaml>.



其后的 `command` 为容器启动时运行的命令, `working_dir` 指定容器启动后的工作目录, `ports` 则把容器的 7051 和 7053 端口分别映射到主机的 7051 和 7053 端口上。

通过 `volumes` 参数可以把宿主机上的目录挂载到容器中, 格式为: “宿主机中的目录: 容器中的目录”, 如 `/var/run/:/host/var/run/` 表示宿主机中的 `/var/run` 目录挂载到了容器中的 `/host/var/run` 目录中。

参数 `depends_on` 定义了 `peer` 容器必须在 `orderer` 容器启动之后才启动, 而 `networks` 则使得容器连接名为 `basic` 的 `docker` 网络中。

### 9.3.3 启动 Fabric 集群

经过 9.3.2 节的准备后, 现已满足所有启动集群的配置要求。下面将使用若干命令来启动集群, 同时还实现创建和加入通道的操作。

1) 使用 `docker-compose` 启动容器:

```
$ docker-compose -f docker-compose.yml up -d ca.example.com orderer.example.com
peer0.org1.example.com couchdb
```

2) 创建通道 `mychannel`:

```
$ docker exec -e "CORE_PEER_LOCALMSPID=Org1MSP" -e "CORE_PEER_MSPCONFIGPATH=/
etc/hyperledger/msp/users/Admin@org1.example.com/msp" peer0.org1.example.com peer
channel create -o orderer.example.com:7050 -c mychannel -f /etc/hyperledger/configtx/
channel.tx
```

3) 把 `peer0.org1.example.com` 加入到通道中:

```
$ docker exec -e "CORE_PEER_LOCALMSPID=Org1MSP" -e "CORE_PEER_MSPCONFIGPATH=/
etc/hyperledger/msp/users/Admin@org1.example.com/msp" peer0.org1.example.com peer
channel join -b mychannel.block
```

4) 查看容器状态:

```
$ docker ps
```

若终端输出下面内容, 则说明集群已正常启动, 接下来可以进行链码的开发。

```
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
e7ae569858e1 hyperledger/fabric-peer "peer node start" 10 minutes ago Up 10
minutes 0.0.0.0:7051->7051/tcp, 0.0.0.0:7053->7053/tcp peer0.org1.example.com
1b9c0c35ead1 hyperledger/fabric-ca "sh -c 'fabric-ca-..." 10 minutes ago Up 10
minutes 0.0.0.0:7054->7054/tcp ca.example.com
0bd7188c7cd3 hyperledger/fabric-orderer "orderer" 10 minutes ago Up 10 minutes
0.0.0.0:7050->7050/tcp orderer.example.com
951b47cc8494 hyperledger/fabric-couchdb "tini -- /docker-e..." 10 minutes ago Up
10 minutes 4369/tcp, 9100/tcp, 0.0.0.0:5984->5984/tcp couchdb
```

### 9.3.4 链码设计

下面将在新创建的集群上部署一个车辆登记应用, 并通过链码来处理业务逻辑。在编





写链码的时候，首先需要确定数据结构，即数据的存储规则，然后定义相应的业务逻辑。以本应用为例，车辆登记包括以下几个操作：登记（添加记录）、修改登记信息（修改记录）以及查询登记信息（查找记录）。详细代码请参考 \$GOPATH/src/fabric-samples/chaincode/fabcar/go。

### 1. 定义数据结构

为简化模型，每条数据只记录基本的车辆信息，记录的定义如下：

```
type Car struct {
    Make string 'json:"make"'
    Model string 'json:"model"'
    Colour string 'json:"colour"'
    Owner string 'json:"owner"'
}
```

结构体 Car 中有 4 个属性，意义如下：

- Make——车辆的厂商；
- Model——车辆的型号；
- Colour——车辆的颜色；
- Owner——车主。

### 2. 实现登记功能

实现登记功能的函数先接收 5 个参数，参数必须按照车辆编号、厂商、型号、颜色、车主的顺序传入，然后根据这些参数实例化一个 Car 数据结构。接着把数据结构序列化成为 json 格式，并用车辆编号作为键（key）存入到账本中。具体代码如下：

```
func (s *SmartContract) createCar(APIStub shim.ChaincodeStubInterface, args []string) sc.Response {
    if len(args) != 5 {
        return shim.Error("Incorrect number of arguments. Expecting 5")
    }
    var car = Car{Make: args[1], Model: args[2], Colour: args[3], Owner: args[4]}
    carAsBytes, _ := json.Marshal(car)
    APIStub.PutState(args[0], carAsBytes)
    return shim.Success(nil)
}
```

### 3. 实现修改功能

由于车辆信息基本不变，因此实现修改登记信息的函数只支持修改车主信息。该函数接收 2 个参数，参数按照车辆编号、新车主的顺序传入。接着该函数根据传入的车辆编号到账本中查找记录，然后把记录的车主更新为新车主。具体代码如下：

```
func (s *SmartContract) changeCarOwner(APIStub shim.ChaincodeStubInterface, args []string) sc.Response {
```

```

if len(args) != 2 {
    return shim.Error("Incorrect number of arguments. Expecting 2")
}
carAsBytes, _ := APIStub.GetState(args[0])
car := Car{}
json.Unmarshal(carAsBytes, &car)
car.Owner = args[1]
carAsBytes, _ = json.Marshal(car)
APIStub.PutState(args[0], carAsBytes)
return shim.Success(nil)
}

```

#### 4. 实现查询功能

查询函数接收车辆编号作为参数，然后返回账本中关于该编号的车辆信息。具体代码如下：

```

func (s *SmartContract) queryCar(APIStub shim.ChaincodeStubInterface, args []string) sc.Response {
    if len(args) != 1 {
        return shim.Error("Incorrect number of arguments. Expecting 1")
    }
    carAsBytes, _ := APIStub.GetState(args[0])
    return shim.Success(carAsBytes)
}

```

#### 5. 实现 Invoke 接口

由于前面实现了多个函数，而 Invoke 接口只能实现一次，因此 Invoke 接口就承担起了转发的职责——它会根据传入的 function 参数决定调用哪个函数。具体代码如下：

```

func (s *SmartContract) Invoke(APIStub shim.ChaincodeStubInterface) sc.Response {
    function, args := APIStub.GetFunctionAndParameters()
    // 根据 function 参数值调用相应的处理函数
    if function == "queryCar" { //queryCar 方法根据传入的车辆 ID 返回车辆信息
        return s.queryCar(APIStub, args)
    } else if function == "initLedger" { // initLedger 函数用于初始化账本的车辆信息，
        // 预先创建了 CAR0~CAR9 共 10 辆车
        return s.initLedger(APIStub)
    } else if function == "createCar" { //createCar 函数用于登记新的车辆信息
        return s.createCar(APIStub, args)
    } else if function == "queryAllCars" { //queryAllCars 函数用于返回所有车辆信息
        return s.queryAllCars(APIStub)
    } else if function == "changeCarOwner" { // changeCarOwner 函数用于修改已登记车辆
        // 的车主
        return s.changeCarOwner(APIStub, args)
    }

    return shim.Error("Invalid Smart Contract function name.")
}

```



### 9.3.5 链码部署

链码编写完成后，接下来进行部署。由于在 peer 节点上安装链码需要 peer 的管理员权限，因此先创建一个 CLI 容器作为 peer 的管理客户端，向 peer 节点发送安装链码、实例化链码等操作。

#### 1) 创建 CLI 容器：

```
$ cd $GOPATH/src/fabric-samples/basic-network
$ docker-compose -f ./docker-compose.yml up -d cli
```

若使用 docker ps 命令返回类似以下的结果，则说明 CLI 容器已成功创建：

```
24639518d32c hyperledger/fabric-tools "/bin/bash" 13 minutes ago Up 13 minutes
cli
```

#### 2) 进入 CLI 容器的命令行环境：

```
$ docker exec -it cli bash
```

#### 3) 安装链码：

```
$ peer chaincode install -n fabcar -v 1.0 -p github.com/fabcar/go
```

#### 4) 实例化链码：

```
$ peer chaincode instantiate -o orderer.example.com:7050 -C mychannel -n fabcar
-v 1.0 -c '{"Args":[""]}' -P "OR ('Org1MSP.member','Org2MSP.member')"
```

#### 5) 执行 Invoke 交易，初始化账本：

```
$ peer chaincode invoke -o orderer.example.com:7050 -C mychannel -n fabcar -c
'{"function":"initLedger","Args":[""]}'
```

#### 6) 查询账本状态：

```
$ peer chaincode invoke -o orderer.example.com:7050 -C mychannel -n fabcar -c
'{"function":"queryCar","Args":["CAR1"]}'
```

查询账本命令将返回以下结果：

```
response:<status:200 message:"OK" payload:"{"colour":"red","make":"Ford",\
"model":"Mustang","owner":"Brad"}"
```

此时链码已成功部署。下面将开发 API 以及前端界面以构建完整应用。

### 9.3.6 SDK 简介

开发应用需要访问账本数据时，需要借助官方 SDK (Software Development Kit) 开发包。SDK 分为 Node.js、Java、Go 等版本，其中 Node.js 和 Java 版本功能和文档最为完善。下面简要分析 Node-SDK 的功能。

### 1. 注册新用户

在 9.3.3 节的集群中，启动 CA 的目的是给新加入的用户签发证书，新用户凭借该证书能够在区块链网络中进行交易。以用户 Alice 为例说明加入网络的流程。

1) Alice 先向 CA 管理员申请加入网络。

2) CA 管理员给 Alice 进行注册，并把注册 ID 和注册密钥返回给 Alice，同时这些信息也存入到 CA 的数据库中。其中 ID 为 Alice 在网络中的标识。

3) Alice 在本地生成公私钥对，同时生成对应的 csr (certificate signing request)，把 csr 连同管理员返回的注册 ID 和注册密钥一同发送给 CA。

4) CA 接收到 csr、注册 ID 和注册密钥后会与数据库中的记录做对比，验证无误后响应 csr 并返回证书给 Alice。

5) Alice 通过 CA 返回的证书在网络中进行交易。

注册新用户的完整代码请参考 \$GOPATH/src/fabric-samples/fabcar 目录下的两个源代码文件 enrollAdmin.js 和 registerUser.js，其中 enrollAdmin.js 用于获得 CA 管理员权限，registerUser.js 用于注册新用户。

由于 CA 服务在启动的时候已经创建了用户 admin 作为默认管理员用户，代码中需先向 CA 获取 admin 用户的私钥并保存在本地。获取 CA 管理员权限的部分代码如下：

```
fabric_ca_client.enroll({
    enrollmentID: 'admin',
    enrollmentSecret: 'adminpw'
}).then((enrollment) => {
    console.log('Successfully enrolled admin user "admin"');
    return fabric_client.createUser(
        {username: 'admin',
         mspid: 'Org1MSP',
         cryptoContent: { privateKeyPEM: enrollment.key.toBytes(),
          signedCertPEM: enrollment.certificate }
        });
}).then((user) => {
    admin_user = user;
    return fabric_client.setUserContext(admin_user);
}).catch((err) => {
    console.error('Failed to enroll and persist admin. Error: ' + err.stack
        ? err.stack : err);
    throw new Error('Failed to enroll admin');
});
```

注册新用户时，先从本地文件中载入管理员用户 admin，然后使用 admin 用户执行注册操作 (register)，接着使用注册过的 ID 和返回的密钥 (secret) 进行登录 (enrollment)，最后把新创建的用户信息保存到本地以作后续使用。注册新用户的部分代码如下 (新注册的用户 ID 为 user1)：

```
fabric_client.getUserContext('admin', true);
```



```

fabric_ca_client.register({enrollmentID: 'user1', affiliation: 'org1.
                             department1'}, admin_user);
}).then((secret) => {
    return fabric_ca_client.enroll({enrollmentID: 'user1', enrollmentSecret:
                                    secret});
}).then((enrollment) => {
    return fabric_client.createUser(
        {username: 'user1',
         mspid: 'Org1MSP',
         cryptoContent: { privateKeyPEM: enrollment.key.toBytes(), signedCertPEM:
                         enrollment.certificate }});
}).then((user) => {
    member_user = user;
    return fabric_client.setUserContext(member_user);
})

```

## 2. 通过新成员发送查询交易

经过上面的操作，用户 user1 已经保存到本地，接下来将通过用户 user1 向网络发送查询交易。完整代码请参考 \$GOPATH/src/fabric-samples/fabcar 目录下的文件 query.js。交易具体流程如下。

### (1) 从本地文件中载入 user1 证书与私钥

用户证书存放路径保存在变量 store\_path 中，程序需要先从目录中载入用户信息再进行下一步操作。部分代码如下：

```

Fabric_Client.newDefaultKeyValueStore({ path: store_path
}).then((state_store) => {
    fabric_client.setStateStore(state_store);
    var crypto_suite = Fabric_Client.newCryptoSuite();
    var crypto_store = Fabric_Client.newCryptoKeyStore({path: store_path});
    crypto_suite.setCryptoKeyStore(crypto_store);
    fabric_client.setCryptoSuite(crypto_suite);
    return fabric_client.setUserContext('user1', true);
})

```

### (2) 构造并发送交易

由于交易需要通过 channel 类发送，所以第一步先实例化 channel，同时把响应交易的 peer 添加到 channel 中。代码如下：

```

var channel = fabric_client.newChannel('mychannel');
var peer = fabric_client.newPeer('grpc://localhost:7051');
channel.addPeer(peer);

```

接下来构造交易并发送。代码如下：

```

const request = {
    chaincodeId: 'fabcar',
    fcn: 'queryAllCars',
    args: ['']
}

```

```
};
return channel.queryByChaincode(request);
```

### (3) 显示返回结果

下面的代码把交易返回结果存放在变量 `query_responses` 中, 并根据其内容打印相应的消息到终端。

```
if (query_responses && query_responses.length == 1) {
    if (query_responses[0] instanceof Error) {
        console.error("error from query = ", query_responses[0]);
    } else {
        console.log("Response is ", query_responses[0].toString());
    }
} else {
    console.log("No payloads were returned from query");
}
```

## 9.3.7 应用的 API 开发

在初步了解了 `$GOPATH/src/fabric-samples/fabcar` 目录下几个 `js` 文件的作用后, 接下来将修改官方示例代码源文件 `invoke.js` 和 `query.js`, 以便在后面构建服务中使用。

在进行后续操作之前, 要确保运行环境中已安装 `node(v6.11.0 或以上)` 和 `npm(3.10.10 或以上)` 工具。

### 1. 安装 npm 依赖包

由于 `$GOPATH/src/fabric-samples/fabcar` 目录下的 `js` 文件都对外部库有依赖, 因此运行 `js` 文件之前需要在本地安装这些依赖包。执行以下命令进入 `$GOPATH/src/fabric-samples/fabcar` 目录并安装依赖包:

```
$ cd $GOPATH/src/fabric-samples/fabcar
$ npm install
```

### 2. 注册新用户

注册新用户需要管理员权限。先执行以下命令获得管理员的证书和私钥:

```
$ node enrollAdmin.js
```

执行完毕后会在当前目录的 `hfc-key-store` 目录下保存管理员的信息。接着执行以下命令注册并登录新用户:

```
$ node registerUser.js
```

同样, 以上命令执行完毕后会在新用户的证书等文件保存在 `hfc-key-store` 目录下。

### 3. 测试查询交易

运行以下命令执行查询交易:



```
$ node query.js
```

该交易调用了链码 Fabriccar 中的 queryAllCars 方法。若终端输出以下信息则说明 query.js 文件正常工作:

```
Successfully loaded user1 from persistence
Query has completed, checking results
Response is [{"Key":"CAR0", "Record":{"colour":"blue","make":"Toyota","model":"Prius", "owner":"Tomoko"}},...
```

#### 4. 链码查询

以下程序在官方示例代码 query.js 的基础上做了修改。修改后的文件名为 queryExport.js, 主要把 query.js 的操作封装在 queryCAR 函数中, 并把该函数暴露以便外部调用, 同时还修改发送交易的内容, 先前的交易调用了链码的 queryAllCars 方法, 而新的交易将调用链码的 queryCar 方法。该交易接收一个参数作为要查询车辆的 ID, 而参数将从界面中输入。下面给出主要代码, 修改过的部分用粗体表示, 完整代码请参考 \$GOPATH/src/fabric-samples/fabcar 目录下的文件 queryExport.js。

```
function queryCAR(car) {
    return Fabric_Client.newDefaultKeyValueStore({ path: store_path
    }).then((state_store) => {
        fabric_client.setStateStore(state_store);
        var crypto_suite = Fabric_Client.newCryptoSuite();
        var crypto_store = Fabric_Client.newCryptoKeyStore({path: store_path});
        crypto_suite.setCryptoKeyStore(crypto_store);
        fabric_client.setCryptoSuite(crypto_suite);
        return fabric_client.getUserContext('user1', true);
    }).then((user_from_store) => {
        if (user_from_store && user_from_store.isEnrolled()) {
            console.log('Successfully loaded user1 from persistence');
            member_user = user_from_store;
        } else {
            throw new Error('Failed to get user1.... run registerUser.js');
        }

        const request = {
            chaincodeId: 'fabcar',
            fcn: 'queryCar',
            args: [car]
        };

        return channel.queryByChaincode(request);
    }).then((query_responses) => {
        console.log("Query has completed, checking results");
        if (query_responses && query_responses.length == 1) {
            if (query_responses[0] instanceof Error) {
                console.error("error from query = ", query_responses[0]);
            }
        }
    })
}
```

```

    } else {
        console.log('responses: ', query_responses[0].toString());
        return query_responses[0].toString();
    }
} else {
    console.log("No payloads were returned from query");
}
}).catch((err) => {
    console.error('Failed to query successfully :: ' + err);
});
}
module.exports.queryCAR = queryCAR;

```

## 5. 链码调用

以下代码也是基于原来官方的 `invoke.js`，经过修改后的新代码在 `invokeExport.js` 文件中，把调用链码的整个流程（包括从生成交易到向排序服务节点、发送交易的背书结果）封装在 `invokecc` 函数里面。`invokecc` 函数接收 2 个参数：`func` 从界面读取，它决定调用链码的具体方法；`arg` 也从界面读取，它是调用链码 `func` 函数时的参数。主要代码如下，其中粗字体表示修改部分，完整代码请参考 `$GOPATH/src/fabric-samples/fabcar` 目录下的文件 `invokeExport.js`。

```

function invokecc(func, arg){
    return Fabric_Client.newDefaultKeyValueStore({ path: store_path
}).then((state_store) => {
    fabric_client.setStateStore(state_store);
    var crypto_suite = Fabric_Client.newCryptoSuite();
    var crypto_store = Fabric_Client.newCryptoKeyStore({path: store_path});
    crypto_suite.setCryptoKeyStore(crypto_store);
    fabric_client.setCryptoSuite(crypto_suite);

    return fabric_client.getUserContext('user1', true);
}).then((user_from_store) => {
    if (user_from_store && user_from_store.isEnrolled()) {
        console.log('Successfully loaded user1 from persistence');
        member_user = user_from_store;
    } else {
        throw new Error('Failed to get user1.... run registerUser.js');
    }

    tx_id = fabric_client.newTransactionID();
    console.log("Assigning transaction_id: ", tx_id.transaction_id);

    var request = {
        chaincodeId: 'fabcar',
        fcn: func,
        args: arg,
        chainId: 'mychannel',

```



```

    txId: tx_id
  };

  return channel.sendTransactionProposal(request);
}).then((results) => {
  var proposalResponses = results[0];
  var proposal = results[1];
  let isProposalGood = false;
  if (proposalResponses && proposalResponses[0].response &&
    proposalResponses[0].response.status === 200) {
    isProposalGood = true;
    console.log('Transaction proposal was good');
  } else {
    console.error('Transaction proposal was bad');
  }
  if (isProposalGood) {
    console.log(util.format(
      'Successfully sent Proposal and received ProposalResponse: Status
      - %s, message - "%s"',
      proposalResponses[0].response.status, proposalResponses[0].
response.message));

    var request = {
      proposalResponses: proposalResponses,
      proposal: proposal
    };

    module.exports.invokecc = invokecc;
  }
});

```

### 9.3.8 界面开发

在 9.3.4 节中，链码中具有代表性的 3 个方法分别为：查找登记车辆消息（queryCar），创建新的车辆信息（createCar）以及修改车辆信息（changeCarOwner）。下面将为这些方法开发简单的用户 Web 页面。每个页面只包含一个表单以及一个提交按钮，表单根据方法的不同会略有差异。

#### 1. 查找登记车辆信息的页面开发

queryCar.html 是查找登记车辆的 HTML 文件，页面信息较为简单。由于该方法只接收一个参数作为其查询车辆信息的依据，因而表单也只有一项输入。具体代码如下：

```

<html>
  <body>
    <head>
      <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
    </head>
    <p> 查询车辆信息 </p>
    <form action="/queryCar" method="post">

```

```

        车辆 ID: <input value="" type="text" name="car"></input>
        <input type="submit" value=" 确定 " ></input>
    </form>
</body>
</html>

```

根据上述代码,当用户按下“确定”按钮时,表单将会发送到 <http://localhost:8080/queryCar>。该地址是一个应用控制器,主要起路由的作用,详细介绍请参考 9.3.9 节。页面的效果如图 9-7 所示。

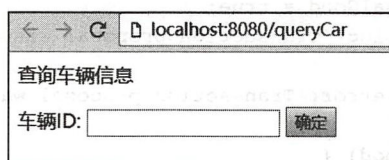


图 9-7 查询车辆信息页面

## 2. 创建新车辆信息的页面开发

创建登记车辆的页面为 `createCar.html`, 代码如下:

```

<html>
    <body>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
    </head>
    <p> 新建车辆 </p>
    <form action="/invoke" method="post">
        <table>
            <tr>
                <th> 属性 </th>
                <th> 值 </th>
            </tr>
            <tr>
                <td> 车辆 ID: </td>
                <td> <input value="" type="text" name="carID" ></input>
                <input type="hidden" name="func" value="createCar">
            </td>
            <td> 厂商: </td>
                <td> <input value="" type="text" name="make" ></input>
            </td>
            <td> 型号: </td>
                <td> <input value="" type="text" name="module" ></input>
            </td>
        </tr>
        </tr>
    </table>
    </form>

```



```

</tr>
<tr>
  <td> 颜色 : </td>
  <td> <input value="" type="text" name="colour" ></input>
</td>
</tr>
<tr>
  <td> 车主 : </td>
  <td> <input value="" type="text" name="owner" ></input>
</td>
</tr>

<tr> <td><input type="submit" value=" 确定 " ></input> </td></tr>
</table>
</form>
</body>
</html>

```

值得注意的是, 表单中的每个输入框都有唯一的标识 ID, 应用控制器需要根据这些 ID 来获取相应的表单信息。效果如图 9-8 所示。

图 9-8 登记车辆信息页面

### 3. 更改车辆信息页面开发

更改车辆信息页面为 changeOwner.html, 代码如下:

```

<html>
  <body>
    <head>
      <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
    </head>
    <p> 更改车主信息 </p>
    <form action="/invoke" method="post">
      <input type="hidden" name="func" value="changeCarOwner"></input>
      车辆 ID: <input value="" type="text" name="carID" ></input>
      &nbsp;
      新车主: <input value="" type="text" name="newOwner" ></input>
    </form>
  </body>
</html>

```

```

        <input type="submit" value=" 确定 " ></input>
    </form>
</body>
</html>

```

页面效果如图 9-9 所示。

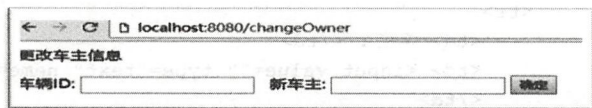


图 9-9 修改车辆信息页面

### 9.3.9 集成

到目前为止，界面以及 API 都已经开发完毕，现在需要设计一个应用控制器，接收来自界面的数据，并根据数据调用相应的 API。完整代码如下（server.js）：

```

var query = require("./queryExport.js");
var invoke = require("./invokeExport.js");
var express = require("express");
var fs = require("fs");
var app = express();
var bodyParser = require("body-parser");
app.use(bodyParser.urlencoded({
    extended: true
}));

app.get('/queryCar', function(request, response) {
    response.writeHead(200, {"Content-Type": "text/html"});
    fs.readFile("html/queryCar.html", "utf-8", function(e, data){
        response.write(data);
        response.end();
    });
});

app.get('/createCar', function(request, response) {
    response.writeHead(200, {"Content-Type": "text/html"});
    fs.readFile("html/createCar.html", "utf-8", function(e, data){
        response.write(data);
        response.end();
    });
});

app.get('/changeOwner', function(request, response) {
    response.writeHead(200, {"Content-Type": "text/html"});
    fs.readFile("html/changeOwner.html", "utf-8", function(e, data){
        response.write(data);
        response.end();
    });
});

```



```

});

app.post('/queryCar', function(request, response) {
  car = request.body.car;
  query.queryCAR(car)
    .then((result) => {
      response.writeHead(200, {'Content-type': 'application/json'});
      if (result.length == 0){
        result = "car not found!"
      }
      response.write(result);
      response.end();
    });
});

app.post('/invoke', function(request, response) {
  func = request.body.func
  console.log(func)

  if (func == 'createCar'){
    carID = request.body.carID;
    make = request.body.make;
    module = request.body.module;
    colour = request.body.colour;
    owner = request.body.owner;
    invoke.invokecc(func, [carID, make, module, colour, owner])
      .then((result) => {
        response.writeHead(200, {'Content-type': 'application/json'});
        response.write(result);
        response.end();
      });
  } else if(func == 'changeCarOwner'){
    carID = request.body.carID;
    owner = request.body.newOwner;
    invoke.invokecc(func, [carID, owner])
      .then((result) => {
        response.writeHead(200, {'Content-type': 'application/json'});
        response.write(result);
        response.end();
      });
  }
});
app.listen(8080)

```

代码使用了 express 模板。它根据 POST 请求的路径来区别操作类型，然后提取表单信息并把信息传进相应的 API 中，最后把 API 返回的响应输出到页面。

### 9.3.10 测试应用

至此，完整的车辆登记应用开发完毕。下面将测试各个页面的功能，检查代码是否如期工作。

### (1) 查找车辆信息

测试查找登记 ID 为“CAR5”的车辆信息，点击确认按钮后会在页面上展示车辆的信息，如图 9-10 和图 9-11 所示。

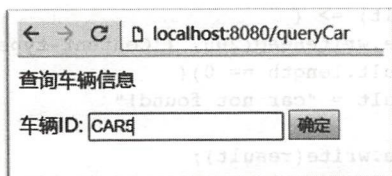


图 9-10 查看车辆信息

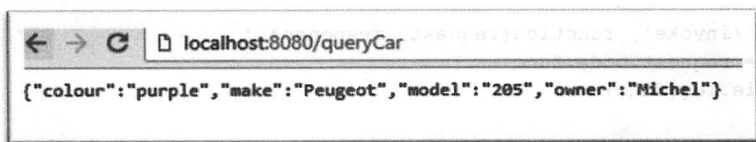


图 9-11 车辆查询结果

### (2) 登记车辆信息 (见图 9-12 至图 9-16)

下面将添加车辆 ID 为“CAR10”的车辆信息。首先查找“CAR10”的相关信息。由于节点的记录中不存在该信息，因此页面返回找不到车辆信息的通知。随后通过添加页面添加“CAR10”的车辆信息，在返回成功添加后再次查询“CAR10”则能成功查询到该车信息。

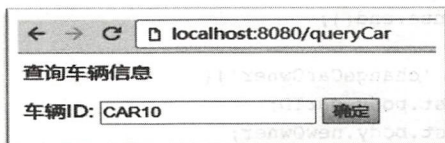


图 9-12 查询无效的车辆信息“CAR10”

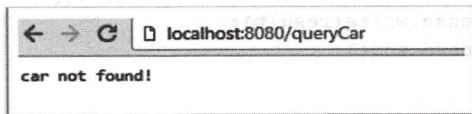


图 9-13 返回无效结果

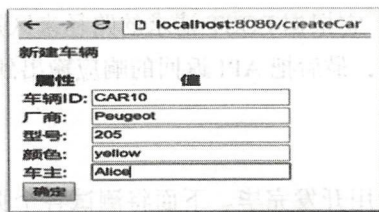


图 9-14 登记新车辆信息



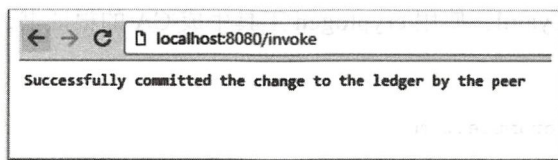


图 9-15 成功登记车辆信息

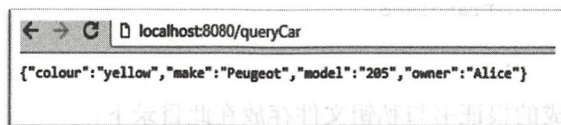


图 9-16 再次查询“CAR10”时返回的结果

### (3) 修改车辆信息 (见图 9-17 和图 9-18)

在前面添加的“CAR10”中, 车辆的“owner”属性为“Alice”, 可以通过修改车辆信息页面把“CAR10”的“owner”属性改为“Bob”。

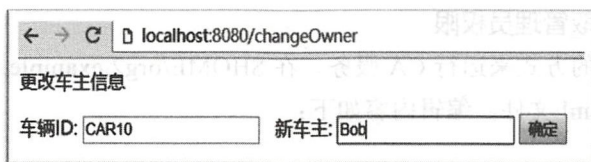


图 9-17 修改车辆信息

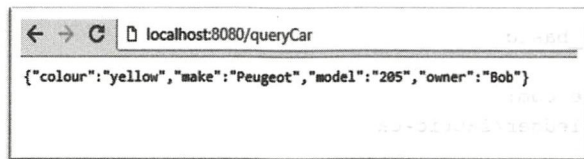


图 9-18 再次查询“CAR10”的返回结果

### 9.3.11 扩展应用中的组织数目

通过前面的实践, 现已基于 Fabric 上开发了一个简单的车辆登记应用。该应用包含了页面、中间服务器等部件, 总体上看来是一个较为完整的应用。然而在原来的设计中, 应用通道只包含了一个组织, 这对后续想参与进来的组织和用户带来了一定的限制, 因此接下来将介绍如何在通道中添加新组织。在此之前, 先了解一下如何创建一个新组织。下面将从创建新组织的 CA 开始, 介绍如何向 CA 申请节点证书、节点的启动以及在节点上安装链码。

#### 1. 生成 CA 根证书

假定新组织命名为 org2.example.com, 创建一个新目录存放生成文件:

```
$ mkdir $HOME/org2.example.com && cd $HOME/org2.example.com
```

编辑 `crypto-config.yaml`, 使用 `cryptogen` 工具生成 CA 的根证书, 内容如下:

```
PeerOrgs:
- Name: Org2
  Domain: org2.example.com
  CA:
    Country: US
    Province: California
    Locality: San Francisco
  Users:
    Count: 1
```

通过 `cryptogen` 生成的根证书与私钥文件存放在此目录下:

```
$HOME/org2.example.com/peerOrganizations/org2.example.com/ca/
```

其中 `ca.org2.example.com-cert.pem` 为 CA 的根证书文件, 另一个以 `_sk` 结尾的文件为 CA 的私钥。由于每次生成的私钥都不相同, 因此私钥的名称会有变化。私钥文件的名称较长, 可以将其重命名为 `ca.org2.example.com-key.pem`。

## 2. 启动 CA 并获取管理员权限

下面将通过容器的方式来运行 CA 服务。在 `$HOME/org2.example.com` 目录下创建并编辑 `docker-compose.yaml` 文件。编辑内容如下:

```
version: '2'
networks:
  default:
    external:
      name: net_basic
services:
  ca.org2.example.com:
    image: hyperledger/fabric-ca
    environment:
      - FABRIC_CA_HOME=/etc/hyperledger/fabric-ca-server
      - FABRIC_CA_SERVER_CA_NAME=ca.org2.example.com
      - FABRIC_CA_SERVER_CA_CERTFILE=/etc/hyperledger/fabric-ca-server-config/
        ca.org2.example.com-cert.pem
      - FABRIC_CA_SERVER_CA_KEYFILE=/etc/hyperledger/fabric-ca-server-config/
        ca.org2.example.com-key.pem
    ports:
      - "8054:7054"
    command: sh -c 'fabric-ca-server start -b admin:adminpw -d'
    volumes:
      - ./crypto-config/peerOrganizations/org2.example.com/ca/:/etc/hyperledger/fabric-
        ca-server-config
    container_name: ca.org2.example.com
    networks:
      - default
```

以上内容编辑完毕后, 通过下面的命令启动 CA 服务:



```
$ docker-compose up -d ca.org2.example.com
```

如果服务正常启动, 则通过 `docker ps` 命令可以看到以下信息:

```
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
6eade3655a5 hyperledger/fabric-ca "sh -c 'fabric-ca-..." 33 seconds ago Up 31
seconds 0.0.0.0:8054->7054/tcp ca.org2.example.com
```

服务启动完毕后, 还需要登录 CA 以获得管理员权限:

```
$ export FABRIC_CA_CLIENT_HOME=$HOME/org2.example.com/client/admin
$ fabric-ca-client enroll -u http://admin:adminpw@localhost:8054
```

若终端输出以下信息, 说明成功获得了管理员的证书与私钥文件, 并将上述文件保存在 `$HOME/org2.example.com/client/admin/msp` 目录下:

```
User provided config file: /home/luke/org2.example.com/client/admin/fabric-ca-
client-config.yaml
generating key: &{A:ecdsa S:256}
encoded CSR
Stored client certificate at $HOME/org2.example.com/client/admin/msp/signcerts/
cert.pem
Stored CA root certificate at $HOME/org2.example.com/client/admin/msp/cacerts/
localhost-8054.pem
```

### 3. 获取节点证书

首先通过 admin 注册 peer0 用户:

```
$ export FABRIC_CA_CLIENT_HOME=$HOME/org2.example.com/client/admin
$ fabric-ca-client register --id.name peer0 --id.type peer --id.affiliation org2
--id.secret peer0pw
```

登记 peer0 并获取相应证书:

```
$ export FABRIC_CA_CLIENT_HOME=$HOME/org2.example.com/client/peer0
$ fabric-ca-client enroll -u http://admin:adminpw@localhost:8054
```

启动新节点时, `$HOME/org2.example.com/client/peer/msp` 将作为新节点的 MSP 目录。由于 CA Server 返回的 msp 目录中没有管理员证书目录, 因此需要创建该目录, 并把前面通过 `cryptogen` 工具生成的 admin 用户证书拷贝到该目录下:

```
$ cd $HOME/org2.example.com
$ mkdir client/peer0/msp/admincerts
$ cp crypto-config/peerOrganizations/org2.example.com/users/Admin@org2.example.
com/msp/signcerts/Admin@org2.example.com-cert.pem client/peer0/msp/admincerts/
```

### 4. 编写节点启动文件

下面定义的 `docker-compose.yaml` 文件将创建 2 个容器: 其中一个作为 peer 节点, 它使

用的是前面通过 CA 服务获取的证书；另外一个则作为 CLI 客户端，负责向节点发送加入通道、安装链码等命令。

追加以下内容到 \$HOME/org2.example.com/docker-compose.yaml:

```
cli.org2:
  container_name: cli.org2
  image: hyperledger/fabric-tools
  tty: true
  environment:
    - GOPATH=/opt/gopath
    - CORE_VM_ENDPOINT=unix:///host/var/run/docker.sock
    - CORE_LOGGING_LEVEL=DEBUG
    - CORE_PEER_ID=cli
    - CORE_PEER_ADDRESS=peer0.org2.example.com:7051
    - CORE_PEER_LOCALMSPID=Org2MSP
    - CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org2.example.com/users/Admin@org2.example.com/msp
    - CORE_CHAINCODE_KEEPALIVE=10
  working_dir: /opt/gopath/src/github.com/hyperledger/fabric/peer
  command: /bin/bash
  volumes:
    - /var/run:/host/var/run/
    - ./chaincode:/opt/gopath/src/github.com/

    - ./crypto-config:/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/
      networks:
      - default

peer0.org2.example.com:
  container_name: peer0.org2.example.com
  image: hyperledger/fabric-peer
  environment:
    - CORE_VM_ENDPOINT=unix:///host/var/run/docker.sock
    - CORE_PEER_ID=peer0.org2.example.com
    - CORE_LOGGING_PEER=debug
    - CORE_CHAINCODE_LOGGING_LEVEL=DEBUG
    - CORE_PEER_LOCALMSPID=Org2MSP
    - CORE_PEER_MSPCONFIGPATH=/etc/hyperledger/msp/peer/
    - CORE_PEER_ADDRESS=peer0.org2.example.com:7051
    - CORE_VM_DOCKER_HOSTCONFIG_NETWORKMODE=net_basic
  working_dir: /opt/gopath/src/github.com/hyperledger/fabric
  command: peer node start
  ports:
    - 8051:7051
    - 8053:7053
  volumes:
    - /var/run:/host/var/run/
    - ./client/peer0/msp:/etc/hyperledger/msp/peer
    - ./config:/etc/hyperledger/configtx
  networks:
    - default
```

## 5. 启动节点

由于需要安装链码，因此把 fabric-samples 中的 chaincode 目录拷贝到 \$HOME/org2.example.com 目录下，使 CLI 容器可以访问 chaincode 目录：

```
$ cp -r $GOPATH/src/fabric-samples/chaincode $HOME/org2.example.com
```

完成 docker-compose 启动文件的编写后，运行以下命令启动容器：

```
$ cd $HOME/org2.example.com
$ docker-compose up -d peer0.org2.example.com cli.org2
```

若 docker ps 命令输出如下内容，则表示两个容器成功启动：

```
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
e9667df48762 hyperledger/fabric-peer "peer node start" About a minute ago Up
About a minute 0.0.0.0:8051->7051/tcp, 0.0.0.0:8053->7053/tcp peer0.org2.example.com
753fec76891c hyperledger/fabric-tools "/bin/bash" 3 minutes ago Up 3 minutes
cli.org2
```

此时执行以下命令，向 orderer.example.com 获取 mychannel 的初始区块：

```
$ docker exec cli.org2 peer channel fetch oldest -c mychannel -o orderer.example.
com:7050
```

新组织对 mychannel 没有读写权限，终端会输出以下错误信息：

```
[channelCmd] readBlock -> DEBU 00a Got status:*orderer.DeliverResponse_Status
Error: can't read the block
```

下面介绍如何修改通道配置，使得 org2.example.com 中的成员能够正常执行交易。

## 6. 修改通道配置

按照 9.2.2 节中对修改通道配置的描述，首先需要获取通道的配置区块，然后添加新组织的相关信息到配置区块中。当编辑配置区块的工作完成后，还需要 org1.example.com 的管理员用户对新的配置区块签名并发送到排序节点。

### (1) 启动 configtxlator 服务

由于 orderer 返回配置文件格式为 proto，为了更友好地编辑配置文件，可先用 configtxlator 工具来把 proto 格式文件转化为 JSON 格式，然后通过 jq 等工具对 JSON 格式的文件进行编辑。

安装 jq 同时启动 configtxlator：

```
$ apt-get install jq && configtxlator start
```

终端输出以下消息说明 configtxlator 服务已成功启动，并监听本地的 7059 端口：

```
[configtxlator] startServer -> INFO 001 Serving HTTP requests on 0.0.0.0:7059
```



## (2) 构造 org2.example.com 的更新内容

由于直接编写更新内容十分烦琐，可以先构造一个包含组织 org2.example.com 信息的临时区块，然后把临时区块中关于 org.example.com 的信息提取出来，作为通道 mychannel 的更新内容。由于临时区块的创建需要 orderer 的参与，因此先创建一个临时的 orderer 组织。

进入 \$HOME/org2.example.com 目录，编辑 tmp.yaml 文件，内容如下：

```
OrdererOrgs:
```

```
- Name: Orderer
  Domain: example.com
  Specs:
    - Hostname: orderer
```

使用 cryptogen 创建 orderer 的证书文件：

```
$ cryptogen generate --config=crypto-config-tmp.yaml
```

接下来定义 configtx.yaml 文件，创建包含 org2.example.com 信息的临时区块。configtx.yaml 的定义如下：

```
---
```

```
Profiles:
```

```
  OneOrgOrdererGenesis:
```

```
    Orderer:
```

```
      <<: *OrdererDefaults
```

```
      Organizations:
```

```
        - *OrdererOrg
```

```
      Consortiums:
```

```
        SampleConsortium:
```

```
          Organizations:
```

```
            - *Org2
```

```
      Organizations:
```

```
        - &OrdererOrg
```

```
          Name: OrdererOrg
```

```
          ID: OrdererMSP
```

```
          MSPDir: crypto-config/ordererOrganizations/example.com/msp
```

```
        - &Org2
```

```
          Name: Org2MSP
```

```
          ID: Org2MSP
```

```
          MSPDir: crypto-config/peerOrganizations/org2.example.com/msp
```

```
          AnchorPeers:
```

```
            - Host: peer0.org2.example.com
```

```
              Port: 7051
```

```
Orderer: &OrdererDefaults
```

```
  OrdererType: solo
```

```
  Addresses:
```

```
    - orderer.example.com:7050
```

```

BatchTimeout: 2s
BatchSize:
  MaxMessageCount: 10
  AbsoluteMaxBytes: 99 MB
  PreferredMaxBytes: 512 KB
Kafka:
  Brokers:
    - 127.0.0.1:9092
Organizations:
Application: &ApplicationDefaults
Organizations:

```

生成临时区块:

```
$ configtxgen -profile OneOrgOrdererGenesis -outputBlock genesis.block
```

临时区块 genesis.block 已包含了新组织的信息, 下面还需要把该信息的 JSON 格式提取出来。把 genesis.block 转化为 JSON 格式:

```
$ curl -X POST --data-binary @genesis_block.pb "localhost:7059/protolator/decode/common.Block" > genesis_block.json
```

提取 Org2MSP (org2.example.com 在通道中标识):

```
$ jq .data.data[0].payload.data.config.channel_group.groups.Consortiums.groups.SampleConsortium.groups.Org2MSP genesis_block.json > Org2MSP.json
```

至此已经成功获取需要向 mychannl 的配置区块中添加的内容。该内容保存在 Org2MSP.json 文件内。

### (3) 获取 mychannel 的配置区块

由于下面的操作涉及的文件较多, 先创建一个目录存放临时文件, 然后把上一步生成的 Org2MSP.json 文件移至该目录:

```
$ mkdir $HOME/configBlock && cd $HOME/configBlock
$ cp $HOME/org2.example.com/Org2MSP.json ./
```

进入 CLI 容器并获取配置区块:

```
$ docker exec -it cli bash
$ cli peer channel fetch config -c mychannel -o orderer.example.com:7050
```

把配置区块拷贝至主机的 \$HOME/configBlock/config\_block.pb:

```
$ scp mychannel_config.block root@root@192.168.1.5:~/configBlock/config_block.pb
```

### (4) 更新 mychannel 的配置区块

在获得了 Org2MSP.json 和 config\_block.pb 2 个文件后, 就可以开始通道的更新工作了。具体步骤如下。

1) 转化 config\_block.pb, 并提取 config 内容存为 config.json:

```
$ curl -X POST --data-binary @config_block.pb "localhost:7059/protolator/decode/common.Block" > config_block.json
$ jq .data.data[0].payload.data.config config_block.json > config.json
```

2) 把 Org2MSP 的内容添加到 config.json, 并把更新后的文件存为 updated\_config.json:

```
$ jq .channel_group.groups.Application.groups.Org2MSP="$(cat Org2MSP.json)" config.json > updated_config.json
```

3) 把 config.json 和 updated\_config.json 文件都转化为 pb 格式, 以计算更新变量:

```
$ curl -X POST --data-binary @config.json "localhost:7059/protolator/encode/common.Config" > config.pb
$ curl -X POST --data-binary @updated_config.json "localhost:7059/protolator/encode/common.Config" > updated_config.pb
```

4) 计算更新变量, 并把更新变量存为 config\_updated.pb:

```
$ curl -s -X POST -F channel="mychannel" -F "original=@config.pb" -F "updated=@updated_config.pb" "localhost:7059/configtxlator/compute/update-from-configs" > config_updated.pb
```

5) 转化 config\_updated.pb 为 json 格式:

```
$ curl -X POST --data-binary @config_updated.pb "localhost:7059/protolator/decode/common.ConfigUpdate" > config_updated.json
```

6) 构造 update\_envelop, 并添加 config\_updated.json 的内容:

```
$ echo '{"payload":{"header":{"channel_header":{"channel_id":"mychannel","type":2}},"data":{"config_update":"'$(cat config_updated.json)'"}}}' | jq . > config_update_in_envelope.json
```

7) 转化 update\_envelop:

```
$ curl -X POST --data-binary @config_update_in_envelope.json "localhost:7059/protolator/encode/common.Envelope" > config_update_in_envelope.pb
```

8) 通过 CLI 更新 mychannel 配置:

```
$ docker exec -it cli bash
$ scp root@192.168.1.5:~/configBlock/config_update_in_envelope.pb ./
$ peer channel update -f config_update_in_envelope.pb -o orderer.example.com:7050 -c mychannel
```

至此, 更新 mychannel 配置的操作完毕, 现在 mychannel 中已经有了 org2.example.com 的相关信息, 后面的内容将验证配置是否修改成功。





### (5) 验证

验证部分通过 cli.org2 容器进行。首先在容器中获取 mychannel 的初始区块, 然后把 peer0.org2.example.com 加入到 mychannel, 最后进行相关的链码操作。

1) 进入 cli.org2 容器内的命令行环境:

```
$ docker exec -it cli.org2
```

2) 获取 mychannel 初始区块:

```
$ peer channel fetch oldest -c mychannel -o orderer.example.com:7050
```

若更改配置的操作正确, 以上命令会在当前目录下生成 mychannel\_oldest.block 文件。

3) 加入 mychannel:

```
$ peer channel join -b mychannel_oldest.block
```

4) 安装链码:

```
$ peer chaincode install -p github.com/fabcar/go -n fabcar -v 1.0
```

5) 执行交易:

```
$ peer chaincode query -C mychannel -n fabcar -c '{"Args":["queryCar","CAR0"]}'  
-o orderer.example.com:7050
```

终端输出以下内容则说明 org2.example.com 已成功加入 mychannel:

```
Query Result: {"colour":"blue","make":"Toyota","model":"Prius","owner":"Tomoko"}
```

至此, 动态添加组织工作完成。值得注意的是, 上述操作能够成功的前提是 mychannel 中原来只有一个组织。若 mychannel 中本来就已经存在多个组织, 则无法通过命令行工具进行通道的升级。此时正确的方法是把构造好的更新交易通过 SDK 的 signChannelConfig 方法收集所有成员的签名, 然后通过 updateChannel 方法向排序节点发送带有签名的交易。若排序节点对签名的验证通过, 则执行更新通道操作。

## 9.4 小结

本章主要阐述了超级账本权限链 Fabric 项目的架构原理和应用开发流程。区块链的技术发展一日千里, 涌现出很多侧重点不同的项目, 相比之下, Fabric 的定位是面向企业应用场景的联盟链通用框架, 架构设计采用了模块化和可插拔方式, 增加了灵活性和可扩展性。Fabric 最大的特点是链上用户具有明确的身份, 采用通道方式确保了交易的隐私性和机密性, 系统的吞吐量也比多数公有链高很多, 使得 Fabric 是目前比较适合构建企业级区块链应用的方案。当然, Fabric 也有不足之处, 如 1.0 版本中共识机制不支持 BFT 类型、吞吐量离商业应用需求还有一定的差距等。这些都是在社区的共同努力下今后需要提升和改进的方向。



## 1. 习题

- [1] 请简述交易预案 (proposal)、交易 (transaction)、数据块 (blob) 和区块 (block) 4 者的关系。
- [2] 请从交易、区块和通道 3 个方面简述排序节点在 Fabric 中的作用。
- [3] 用 Go 语言开发链码所需要实现的接口有哪些?
- [4] 下载 Fabric 源码<sup>⊖</sup>并编译 peer 和 orderer 的二进制可执行文件, 然后通过环境变量的方式来配置和启动 peer 和 orderer 进程。在此基础上实现通道创建、安装和实例化链码 chaincode\_example02<sup>⊖</sup>这 3 个操作。

## 2. 参考资料

- [1] 邹均, 张海宁, 唐屹, 李磊等. 区块链技术指南 [M]. 北京: 机械工业出版社, 2016.
- [2] 杨保华, 陈昌. 区块链原理: 设计与应用 [M]. 北京: 机械工业出版社, 2017.
- [3] Hyperledger Fabric Develop Group. Fabric Introduction[EB/OL]. [2017-12-16]. <http://hyperledger-fabric.readthedocs.io/en/latest/blockchain.html>.
- [4] Hyperledger Fabric Develop Group. Hyperledger Fabric SDK for Node.js[EB/OL]. [2018-1-20]. <https://fabric-sdk-node.github.io/index.html>.

---

⊖ <https://github.com/hyperledger/fabric/tree/release-1.0>.

⊖ [https://github.com/hyperledger/fabric/tree/release-1.0/examples/chaincode/go/chaincode\\_example02](https://github.com/hyperledger/fabric/tree/release-1.0/examples/chaincode/go/chaincode_example02).



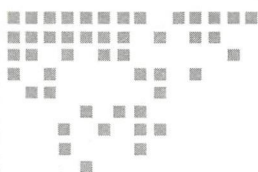
### 第三篇 *Part 3*

## 进阶篇

- 第 10 章 其他区块链平台
- 第 11 章 区块链常见问题剖析
- 第 12 章 区块链评测







## 其他区块链平台

作者：邵周

通过前面章节的学习，我们已经掌握了基础的区块链技术原理和构架，也认识了一些主流的区块链技术平台及其特点。本章我们将分析目前主流区块链平台存在的问题和挑战，研究思考区块链平台的典型需求和发展方向，简单介绍有可能成为区块链 3.0 技术代表的其他区块链平台，以及其特点及应用，目的是博采众长、融会贯通，从而更好地加强我们对区块链技术认知和掌握。

### 10.1 区块链架构存在的问题和挑战

现实世界不可能是完美的，我们往往在各种三元悖论的权衡之中寻找平衡点。在经济学上有个“蒙代尔不可能三角”（Mundellian Trilemma），指的是一个国家不可能把独立货币政策（Monetary Policy）、固定汇率（Exchange Rate）和资本自由流动（Capital Mobility）三者兼顾到，往往只能三取二舍一。在分布式计算领域，也有人们认可并熟知的 CAP 理论，即一致性（Consistence）、可用性（Availability）和分区容错性（Network Partitioning）三者不能得兼，只能三选二，如图 10-1 所示。

而在区块链领域，目前公认的一个“不可能三角”是在去中心化（Decentralization）、安全性（Security）和扩展性（Scalability）上也只能三者选其二，如图 10-2 所示。完全去中心的区块链架构会带来性能和安全性方面的问题。传统的分布式架构，例如云计算，是通过把一个任务切片，分发到多个节点来并行计算，最后将结果汇总回单一节点，因此效率很高，但中心化程度也很高。完全去中心的区块链架构是把同一个任务放在多个节点同时运行，结果最后进行共识，共识的结果成为最终状态。可想而知，其效率会远远低于前者，



但在安全性和可信任程度上又远远高于前者。

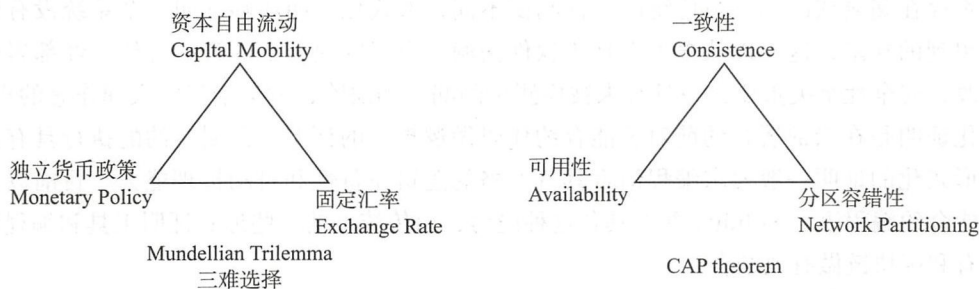


图 10-1 蒙代尔不可能三角和 CAP 理论

根据区块链的不可能三角理论，很明显目前以太坊架构是以牺牲扩展性来换取去中心和安全性的。以太坊上平均每秒大约能做 15 笔交易，所以至今为止，除了 ICO 或数字资产发行这种应用外，以太坊还没有出现上规模的应用。2017 年出现的一个区块链养猫游戏“CryptoKitties”就造成了以太坊网络的拥堵。因此，标榜区块链 3.0 的其他各类区块链平台项目，纷纷针对比特币、以太坊的性能问题提出了自己的改进方案。然而，目前区块链落地应用少，却不只是因为区块链性能上的问题。我们首先来看看当前区块链平台的问题和痛点。

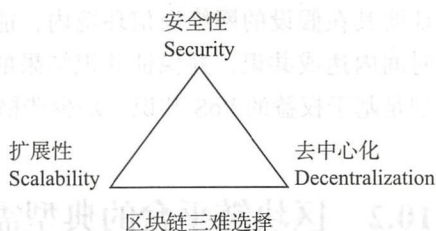


图 10-2 区块链不可能三角理论

1) 交易性能限制。比特币的理论设计限制了其每秒只能完成 7 笔交易，通常每秒只能完成 1~2 笔；以太坊大约每秒处理 10~20 笔交易。另外，PoW 这种最终一致性共识算法，使得如比特币的交易需要获得 6 个以上的确认才能在很大概率上保证交易的最终确定性，因此交易确认的时间也长。

2) 扩展性限制。在需要立即交易确定性的场景，例如联盟链或私有链的场景中，一般节点数会受限。例如使用基于 Paxos 或 BFT（拜占庭容错）等基于状态机复制（SMR）共识算法的区块链平台，参与共识的节点数一般不能超过两位数。

3) 易用性。目前智能合约的开发、部署和调用都需要由专业区块链编程人员进行，区块链在易用性和对应用的支持上还存在较大障碍。

4) 兼容性和互操作性。目前大部分的区块链平台都是独立系统，不支持区块链上的资产在不同链之间的流转，不同链上的应用也没有办法做到跨链的集成和整合。

5) 数据存储。区块链上数据存储容量有限制，特别是每个全节点需要存储的数据量越来越大，急需更低成本和高效的分布式存储方案。

6) 智能合约形式化证明。形式化证明（Formal Verification），出于成本的考虑，以往



往只应用在硬件设计中，后来慢慢被引入软件行业。形式化证明通过数学方法来证明代码中是否存在漏洞或缺陷。与传统的软件测试不同，形式化证明能够证明一个系统没有任何可以想到的缺陷，这可以从根本上杜绝软件漏洞。由于区块链上的智能合约一经部署就不能修改，安全性至关重要，一旦有未被检测出的漏洞和缺陷，就会给黑客大肆作恶的机会。形式化证明是在未部署合约前对智能合约代码做逻辑上的证明，证明合约的执行具有正确性。形式化的证明一般要求编程语言具备严格的逻辑完备性和自动推理能力。目前以太坊的智能合约编程语言 Solidity 并不具备这种能力，而传统上的一些形式证明工具和编程语言还没有和区块链做有效结合。

7) 数据同步性能限制。链上数据每时每刻的增长，带来的问题就是网络上数据同步的负担越来越大，同步速度也越来越慢。

8) 共识机制数学证明。很多区块链平台使用的共识算法并没有给出严格的数学证明，证明其在假设的网络通信环境内，能够在少数故障节点存在的情况下，依然能够在有限的时间内达成共识，并保证共识结果的正确性。特别是在公有链情况下，很多共识算法，特别是基于权益的 PoS 共识，缺少严格的数学证明。

## 10.2 区块链平台的典型需求和发展方向

### 10.2.1 区块链平台的典型需求

区块链上的应用根据场景不同，有以下不同维度的需求。

1) 隐私保护。交易参与方和交易的细节不能泄露给交易无关方。目前比特币、以太坊等只是提供半匿名机制，不能保证在大数据分析下隐私交易信息不被泄露。

2) 链上安全性。在区块链网络，特别是公有链场景下如何容错，特别是拜占庭容错，防止女巫、重放、DDoS 等攻击是一个十分重要的需求。

3) 数据真实性。虽然在区块链上的数据不可篡改，但现实的数据却没有办法防止人为造假。如何保障链下数据真实上链，又不重回到中心化的 Oracle（预言机）机制，也是非常关键的需求。Oracle 是一个连接链下链上的数据真实性保障机制，通常需要由权威性机构来做 Oracle，但如何防止 Oracle 的中心化是未来预言机机制设计的关键考虑点。

4) 有用工作量证明。比特币挖矿造成能源的浪费巨大，有人估算目前比特币一年挖矿所耗费的电量已经相当于爱尔兰一年的用电量。比特币的工作量证明只是不停地做哈希运算，计算结果也没有什么实际用途。如果既能做有用的工作量，又能提供一个比较稳定的共识机制，是非常有吸引力，但也是比较难的研究方向。

5) 密码安全性。量子计算目前虽然还是起步的初始阶段，离发展到商用可能还要一大段距离，但量子计算的 Shor 算法已经在理论上能破解公钥加密系统，区块链所基于的公钥签名系统都将面临被量子计算破解的威胁。另一方面，量子计算的 Grover 算法虽然不能在





数量级上降低破解哈希算法的难度,但也能减少一部分破解哈希算法的计算量。因此,抗量子密码算法也成为目前区块链的一个研究方向。

6) **治理和监管**。很多区块链项目目前都是依靠几个核心开发人员维护。一些关键参数的修改和平台发展方向的决策还是由几个人决定,缺少一个比较透明、严格的治理架构和监管体系。如何设计一套适合于区块链的治理和监管体系,使得区块链平台能健康发展,是当下很多区块链项目面临的一个问题。另一方面,一些采用常规计算的破解现有密码学技术的算法也会被发现,像2017年谷歌成功实现对SHA-1的有效碰撞,也说明即使不考虑量子计算的威胁,密码安全性也是一个需要重点关注的问题。

7) **智能合约升级、监控**。智能合约一经部署就不能更改,但传统软件都需要不断完善和升级。另外在很多场景下,智能合约的逻辑也需要适应需求的变化。因此如何对智能合约升级,以及如何监控智能合约的运行,也是一个比较常见的需求,特别是对企业客户来说。

8) **防止算力中心化**。由于比特币的价格攀升,利益驱使比特币挖矿成为一场算力的军备竞争。比特币挖矿的算力不断趋于中心化,目前五大矿池的算力已经超过全网算力的70%。防止算力集中,目前常用的方法是设计需要存储量大的工作量证明机制,这样在成本上抵抗做专门的算力挖矿芯片(ASIC),使得普通人仍然可以用通用处理器,像CPU、GPU来参与挖矿共识。

9) **身份认证及权限控制**。很多区块链的应用,特别是企业级的联盟链应用,需要确认用户的身份,并根据其身份来授予不同的权限。安全高效的身份ID管理、身份认证和权限控制是一个重要的需求。

10) **移动区块链**:目前区块链系统都是运行在服务器节点上,手机上只是钱包或轻节点。但未来随着移动设备算力以及存储能力的提升,直接在移动设备组成的去中心化网络成为可能,成为真正意义上的移动区块链平台。

## 10.2.2 区块链平台的发展方向

上述对区块链平台的行业需求,驱动着区块链向实用性稳定性和安全性方向发展,而具体的技术创新则是在以下一些方向上拓展。

1) **区块链操作系统**。类似传统电脑操作系统给电脑使用者和应用开发提供底层服务一样,区块链操作系统也给上层去中心化应用的开发和运行提供支撑的功能和环境,这些功能包括认证、权限控制、数据存取、合约编译、测试级部署等支撑性功能。操作系统提供易用性,降低使用区块链以及开发DApp的门槛。

2) **区块链中间件**。和传统中间件一样,区块链中间件屏蔽不同区块链平台的接口差异,使得应用能够方便地开发支持不同区块链的DApp,另外也提供跨链整合功能,使得应用能够集成整合多个区块链平台。

3) **区块链网络(多链)**。和比特币或以太坊的单链技术不一样,区块链网络提供一个网



络平台。在该平台下有多条链，每个链上都可以发行自己的虚拟资产，链与链之间也可以通过区块链网络平台中的主链或中继来实现跨链互连。

4) 侧链技术。侧链最早来自于一个无须改变主链（原指比特币的主链），但又能够通过建立一个侧链来扩展主链功能的概念。主链的资产可以安全转移到侧链上流通，最好也能安全地回到主链。

5) 区块扩容。区块容量和能打包的交易多少有关，由于区块链的出块频率相对固定，因此当区块扩容之后，每秒交易量（TPS）也会相应提高。

6) 链下计算。区块链的链上计算虽然能安全可靠，但是共识开销很大，因此把有些计算移到链下来处理，从而提升交易速度是一个必然的选择。

7) 分区共识、分片。以太坊的性能问题让以太坊的开发社区也意识到，当每个交易、每步智能合约的执行都需要在所有的共识节点去完成的时候，其扩展性是无法支撑应用的。因此，分片处理、采用局部共识代替全局共识不可避免。

8) 区块链存储。区块链上能存储的空间有限，一般来说只是存储数据的哈希值，也就是数据的指纹。因此如何解决区块链数据，特别是代码或状态数据的存储是当今区块链平台都面临的一个问题。

9) 共识机制。区块链最核心的部分是其共识机制。不同的共识机制有着不同的共识属性，同时也决定其每秒交易量，因此共识机制的设计也成为区块链底层平台设计的一个关键环节。

## 10.3 其他区块链平台

区块链技术当前正处于向实用性过渡阶段，新型的区块链项目层出不穷，百花齐放，都在竞争成为下一代区块链技术的典型代表，但大部分还没有正式成型，未来能否实现白皮书中所描述的愿景还有待验证。区块链或者分布式账本技术只能在去中心化、安全性和可扩展性之间三选二。区块链技术的特质决定了去中心化和安全性的重要性，所以目前的区块链平台大多选择确保安全性和去中心化，而在可扩展性方面做了妥协。也有一些项目尽量确保安全性和扩展性，允许稍稍中心化。未来的区块链平台之争，主要是看在特定的应用场景下，哪些平台和技术在这三方面有更好的实现。下面我们简单介绍一些比较典型的、有可能成为未来最具实用化的区块链平台，以及这些平台的特点和应用。

### 10.3.1 以太坊：区块链龙头的转型升级

以太坊是区块链智能合约技术的优秀代表，是目前最成熟、应用最广泛的区块链平台之一。目前区块链应用最大的限制就是延迟和数据吞吐量。以太坊也不例外，不过它自身也在不断迭代发展，通过如 Plasma、State Channel、Raiden、Truebit、Sharding、Casper 等扩容方案，力求将其自身提升至区块链 3.0 的架构。下面给出网址资源，有兴趣的读者可自





行深入研究。

❑ 官网: <https://www.ethereum.org>

❑ 源码: <https://github.com/ethereum>

❑ 白皮书: <https://github.com/ethereum/wiki/wiki/White-Paper>

由于我们在第8章已经对以太坊做了详细的介绍, 这里就不再重复。下面我们仅介绍以太坊今后的升级扩容架构方案。

## 1. Plasma

Plasma 是由 Joseph Poon 和 Vitalik Buterin 提出的区块链扩展解决方案。其思想在很大程度上类似于“闪电网络”的理念, 主要是在主网中冻结资金并将其用于链下的结算, 而交易则在链下进行。在 Plasma 中的主链, 也就是根链是以太坊区块链, 链下交易在基于主链创建的与交易参与者相关的子链上进行。如果交易一方发现对方有不当行为(违反了预先确定的交易协议), 他可以把其违规的证据提交到主网络, 导致作恶的一方被罚款。子链中的共识协议以及参与者的行为和功能的执行都是由主网中的一个智能合约控制的。子链的交易数据采用 Merkle 树来组织, 其默克尔证明被上传记录在主链上, 允许外部观察者监控子链的状态。

Plasma 利用子链来减轻根链(以太坊)负担的方式来提高交易流量, 同时用根链的安全性来保证子链的安全性。Plasma 的扩展性甚至可以达到每秒亿级的交易。这是因为其可以在子链上再建子链, 采用树形结构不断扩展, 交易在各个层次化的子链上并行执行。Plasma 由以下组件组成。

❑ Client (客户端)——监控 Ethereum 且运行子链, 当检测到诈欺的行为就立即离开。

❑ Child chain (子链)——监控在 Ethereum 上 deposit 的行为, 并执行所有与该子链目前状态相关的计算。

❑ Root chain (根链)——透过智能合约锚定子链在 Ethereum 链上, 处理子链上要存款与离开(提款)的业务, 当收到足够的信息进行确认与处理, 或是拒绝诈欺的提款(离开)。

❑ Parent chain (母链)——作用是保护子链, 在最小可行版本(MVP)上母链等同于根链。但在最终版本上, 子链上可能会有多个母链, 但只有一个根链。

## 2. State Channel

状态通道(State Channel)是一种用于执行交易和其他状态更新的链下“off-chain”技术。像闪电网络和雷电网中的支付通道一样, 它也是状态通道的一种, 区别在于支付通道只关注支付状态, 而非其他智能合约中的状态。状态通道不仅可以用于支付, 也可用于区块链上任意的状态更新, 包括智能合约的更改。状态通道通过在区块链上部署一个规则合约, 由该合约来向链上提交最终状态。状态通道最适合有一组明确参与者的应用程序, 特别是在需要长期交换大量状态更新的情况下非常有效。状态通道有很强的隐私属性。因





为一切都发生在参与者之间的通道“内”，而不是公共广播并记录在链上。只有开启和关闭交易必须公开。状态通道的权威性是即时生效的。这意味着只要双方签署了一个状态更新，可以被认为是最终状态。双方都有明确保证，在必要的情况下，他们可以将状态“执行”到链上。

### 3. Raiden

雷电网络（Raiden Network）是一个链外扩展解决方案，是以太坊区块链上的基础设施层。雷电网络目前正在开发中，支持即时转账、低成本、可扩展和保护隐私。雷电网络虽然基本的出发点很简单，但底层协议相当复杂，实现起来也不容易。雷电网络是基于闪电网络思路在以太坊上实现的一个链下支付通道网络。它通过智能合约来实现双方的支付，包括条件支付功能。雷电网络通过向链上合约提交“Update Transaction”报文来实现支付，通过函数调用实现 Smart Condition，可以提供比闪电网络的 HTLC（Hash Time Lock Contract，哈希时间锁定合约）更丰富的条件支付功能。

雷电网络项目又分为三个独立的子项目： $\mu$ Raiden（微雷电）、雷电网络和 Raidos（雷电网络 2.0）。 $\mu$ Raiden 和雷电网络有相同的一些属性。它能在两方间提供无须信任的、即时和免费的转账功能。它是为多对一支付而设计的，像用户和一个 DApp 交互。但它不适合于多对多的支付场合，因为它需要每个潜在的支付方预先锁定一些 token。这个限制主要是为了降低技术的复杂度，可以让微雷电在现在的以太坊主链上使用。

Raidos（或者 Raiden 2.0，dos 是西班牙语的“二”）是一个用侧链技术来实现通用的状态通道。雷电网络只支持 ERC20 的 token 转移，Raidos 旨在将以太坊的通用计算能力扩展到一个卫星链网络，并可以支撑任何智能合约。这种计算和以太坊的分片类似，并能互补。

雷电网络在 2017 年发布了开发者 Preview 版本，计划于 2018 年推出最初版本。

### 4. Truebit

Truebit 项目是把大计算量的工作移到链下。这样可以扩展智能合约的计算能力，而不受以太坊“Gas 限制”的限制。与其他扩容项目思路有些类似，TrueBit 使用区块链上的分层来解决这个问题。但其特殊之处在于将计算的执行和验证都进行外包。链外节点可以执行计算任务，而非链上每一个节点计算每一智能合约。这些参与者被称为“解决者”，他们以提出一个解决问题的方式来获得奖励，而“验证者”检查他们的工作。Truebit 项目开发团队包括网络智能合约语言 Solidity 的创建者 Christian Reitwiessner，因此很多人对此寄予厚望。

### 5. Sharding

当前所有的区块链协议中，每个全节点都保存所有的状态，包括账户余额、智能合约代码以及智能合约的“世界状态”存储。这提供了很高的安全性，但是扩展性大大受限。因为区块链的交易处理速度不能超过其中任何一个节点的处理速度，所以像比特币和以太坊的交易处理速度会很慢，每秒交易不会超过 20 笔。那有什么解决办法呢？根据区块链不

可能三角，只能通过牺牲一部分安全性或去中心化程度来提高扩展性。以太坊的分片正是采用了这一思路。

以太坊目前的分片思路是把状态和历史数据分成  $K$  份，每份是一片（Shard）。例如，一种分片方法可以简单地用地址段来分片， $0x00$  开始的地址为一片， $0x01$  开始的地址为另一片，依此类推。一种简单的分片机制是每一片都有自己的交易历史，交易的效果也只是局限在那一片。这种机制适合于支持多资产的区块链，每种资产放在一个片中。更复杂的分片机制则支持片与片之间通信。在这种机制下，一个片上的交易会触发另一个片上的事件。

以太坊分片的设计者提出了一个简单的分片方案。在区块链上，有一些叫“核证人”（Collator）的特殊节点在分片  $K$  上接收消息，同时建立核证记录（Collation）。每个核证记录都有核证表头（header），核证表头里包括前核证头哈希值以及接收消息的 Merkle 树根。每个片中的核证记录连接成一个像传统区块链的链。在分片机制下，主链仍然存在，但主链的职能只是保存所有分片的核证表头。每个分片中最长的核证记录链是被默认的共识链。

分片目前还只是一个设想，具体实现会遇到很多挑战，其中最大的挑战就是片与片的通信。

## 6. Casper

PoW 共识机制始终会是以太坊性能的瓶颈。因此以太坊最早设计的路线图中，就有未来采用 Casper PoS 共识机制的构想，只是在实现上由于安全等多方面原因不断推迟。目前从 Casper 的白皮书初稿来看，第一阶段的 Casper 会是 PoW 和 PoS 的混合机制。具体来说，将区块按每 100 个做一个最终确认，前 99 个采用 PoW 方式出块，第 100 个采用 PoS 机制出块，同时将前 100 个块的交易数据最终确认，不能再推翻。以太坊的 PoS 机制也借鉴了类似 Tendermint 的 BFT 设计，采用押金和投票方式来保证见证人不能作恶。

利用改进共识机制来提升交易频率的区块链平台还有很多，典型的像 NXT（PoS 共识）、Ripple（用 UNL 局部共识代替全局共识）、Stellar（联邦式 BFT 共识）、Bitshares/EOS DPoS 共识等。

## 10.3.2 EOS：区块链操作系统

EOS 可以在垂直和水平 2 个方向扩展以满足大量分布式的应用需求。这种新的区块链架构方式，类似传统的操作系统，可以向用户提供包括账户系统、用户认证、授权、数据库、异步通信、调度、云存储等功能和服务，用户可方便快捷地免费部署分布式应用。根据 EOS 白皮书显示，其预计扩展至单链几千 TPS、全网并行百万 TPS 的交易吞吐量，并且主网于 2018 年 6 月上线，是未来区块链平台强有力的竞争者。同前面一样，这里给出网址资源，供有兴趣的读者自行进行更深入的研究。

□ 官网：<https://eos.io>



❑ 源码: <https://github.com/eosio>

❑ 白皮书: <https://github.com/EOSIO/Documentation/blob/master/TechnicalWhitePaper.md>

值得一提的是,和 Dan Larimer 之前主导过的 2 个成功项目 BitShares 和 Steem 一样, EOS 采用了由比特股团队 cryptonomex 开发,采用 C++ 编写的石墨烯 (graphene) 技术。此技术并发能力比较强,基于石墨烯底层的几个项目均可以达到 1.5S 的平均确认速度和有限条件下实测 3300TPS 的数据吞吐量,所以 EOS 才有信心在引入并行链的方式后, EOS 网络最高可以达到数百万 TPS,并且本地链可达到毫秒级的确认速度。

## 1. EOS 特性

在共识机制上, EOS 采用了 DPoS 股份授权证明算法,全网持有代币的人可以通过投票来选择区块生产者,一旦当选任何人都可以参与区块的生产。这种设计缩短了区块的确认时间,同时也节省了资源,但是依赖于代币持有者和投票,可能并不完全去中心化,但这算是利用改进共识机制来提升交易吞吐量和确认时间的一个很好的例子。在本章写作时, EOS 发布了第一个功能完备的预发行版——Dawn 3.0,而后在本章审校过程中,陆续发布 Dawn 4.0,直到发布发行版 EOSIO 1.0。可以看出其又做了很多关于扩容和增加安全性的努力和尝试。

❑ 利用并行计算增加吞吐量,从而提高可扩展性。

❑ 在协议设计中加入轻客户端验证机制,轻客户端就可以验证区块链中的交易,而无需处理整个区块链,从而实现跨链通讯。

❑ 设计稀疏区块头验证 (Sparse Header Verification)。轻客户端只需处理两种区块的头部:活跃生产节点变化以及包含相关跨链通讯的区块。这大大减少了轻客户端维护拜占庭容错的开销,并大大提高了跨链通讯的效率。

❑ “上下文无关行为” (Context Free Actions) 是实现高效跨链通讯的关键功能之一,使 EOS 能够并行化与跨链通讯相关的绝大部分算力承载。

❑ 利用交易压缩,区块链可以更有效地存储和传输大量交易。

❑ Dawn 3.0 现在默认使用 Binaryen WebAssembly 解释器,而不是更快的 Just-in-Time (JIT) 编译器。这会降低性能,但会增加稳定性和标准一致性。这是一个很好的安全性妥协。

❑ BIOS 架构,在 EOSIO Dawn 3.0 下,绝大多数区块链业务逻辑已经转变为智能合约,可以由社区动态更新而不需要硬分叉。

❑ Dawn 3.0 最重要的功能之一是增加了用户可配置的延迟以适应不同的操作来完成安全延迟交易。

❑ 每个账户至少有两个权限级别:“owner”和“active”,通过巧妙设计,实现在 EOS 链上丢失密码可恢复。

下面我们看一下 EOS 和 Ethereum 的对比,如表 10-1 所示。



表 10-1 EOS 和 Ethereum 对比

| 对比项    | EOS  | Ethereum   |
|--------|--|--|
| 共识算法   | 委托股权证明机制 (DPOS), 较为中心化   | 工作量证明机制, 计划向 PoS/PoW 混合机制迁移, 较为去中心化                              |
| 容错性    | 损坏程序的修复机制 (冻结和修复损坏或冻结的程序)  | 失败和损坏的程序会造成投资损失, 或者区块链硬分叉 (DAO 程序的失败, 导致了 ETH 和 ETC 的分叉)         |
| 硬分叉风险  | 无多条区块链分叉的风险  | 修复一个损坏的程序, 会以全网为代价, 可能会导致分叉                                      |
| 治理     | 用具有合法约束力的区块链宪法, 建立共同的司法管辖, 链上投票合约实现更高效的链上治理和协议及参数切换                  | 没有区块链宪法, 只是通过以太坊基金会进行社区化治理, 容易产生社区分裂                             |
| 易用性    | 为开发者提供全套的加密和区块链通讯功能, 使得开发者能够专注于商业逻辑的功能开发                             | 以太坊目前有较为完善的生态, 目前有不少开发工具支持, 也有不少合约模板可供使用                         |
| 扩展性    | 通过横向和纵向的扩展, 可以让 EOS 网络处理能力达到每秒处理上百万笔交易                               | Vitalik Buterin 提出了无限扩容路线图, 通过数据库切片、Plasma、状态通道、Trubit 等技术进行横向扩容 |
| 应用支持   | 可以支持数千个工业规模的去中心化应用   | 在扩容实施前, 只能支持小规模的交易量应用  |
| 抗 DDoS | 对单个 Block Producer 的拒绝服务 (DoS) 攻击, 不会让整个网络中断, 但相对固定的 21 个 BP 容易被集中攻击 | 目前暴涨的交易量通常会让整个网络冻结拥堵, 急需进行扩容                                     |
| 交易费用   | 零交易费用, DApp 用户无交易成本, DApp 开发者需要为用户购买 ram 和 cpu 资源                    | 每一次计算, 存储操作和带宽占用, DApp 用户都需要消耗 Gas, 为此支付以太币                      |

## 2. EOS 总体架构

图 10-3 为 EOS 部署架构。EOS 的核心是由若干区块生产节点组成的 EOS 网络, 区块生产者按照投票协议以获得投票数为依据在每个产块轮次内选定 21 个产块节点进行产块。每个产块节点在 3 秒内需要产生 6 个区块, 即严格按照 0.5 秒时间槽产生一个区块, 如果在 0.5 秒时间槽内没有及时产出区块, 这个时间槽就会被跳过。一旦一个生产者错过一次出块, 并在其后的 24 小时内没有一次出块, 生产者将被移除出, 以剔除不可靠的出块者, 保证网络平稳运行, 直到其再次发起申请成为出块者。

EOS 网络由若干的 EOS Block Producer (BP) 节点组成。这些节点在主网上线时将共用一个 genesis.json 配置。genesis.json 是在以太坊 EOS 众筹结束后从以太坊众筹合约中快照生成的配置, 其中记录了 EOS 地址映射后各个地址拥有的 EOS 个数。每一个 BP 节点包括 2 个核心进程。一个是 nodeosd, 提供了 EOS 区块链节点的 P2P 网络、账本、智能合约等核心功能, 其内置的 eosio 合约作为系统合约实现了 DPoS 投票选举 BP 的功能。nodeosd 对外提供 2 个端口: 一个是默认为 8888 的 API 调用端口, 所有对链的操作和查询、对合约的操作和查询都可以通过这个 API 端口实现; 另一个是默认为 9876 的 P2P 端口, 基于这个端口实现同其他产块节点的 P2P 连接, 执行区块同步。一个 BP 节点还包括一个钱包管理进程 keosd, 提供钱包相关的管理工作。EOSIO 默认提供 cleos 命令行工具, 通过连接

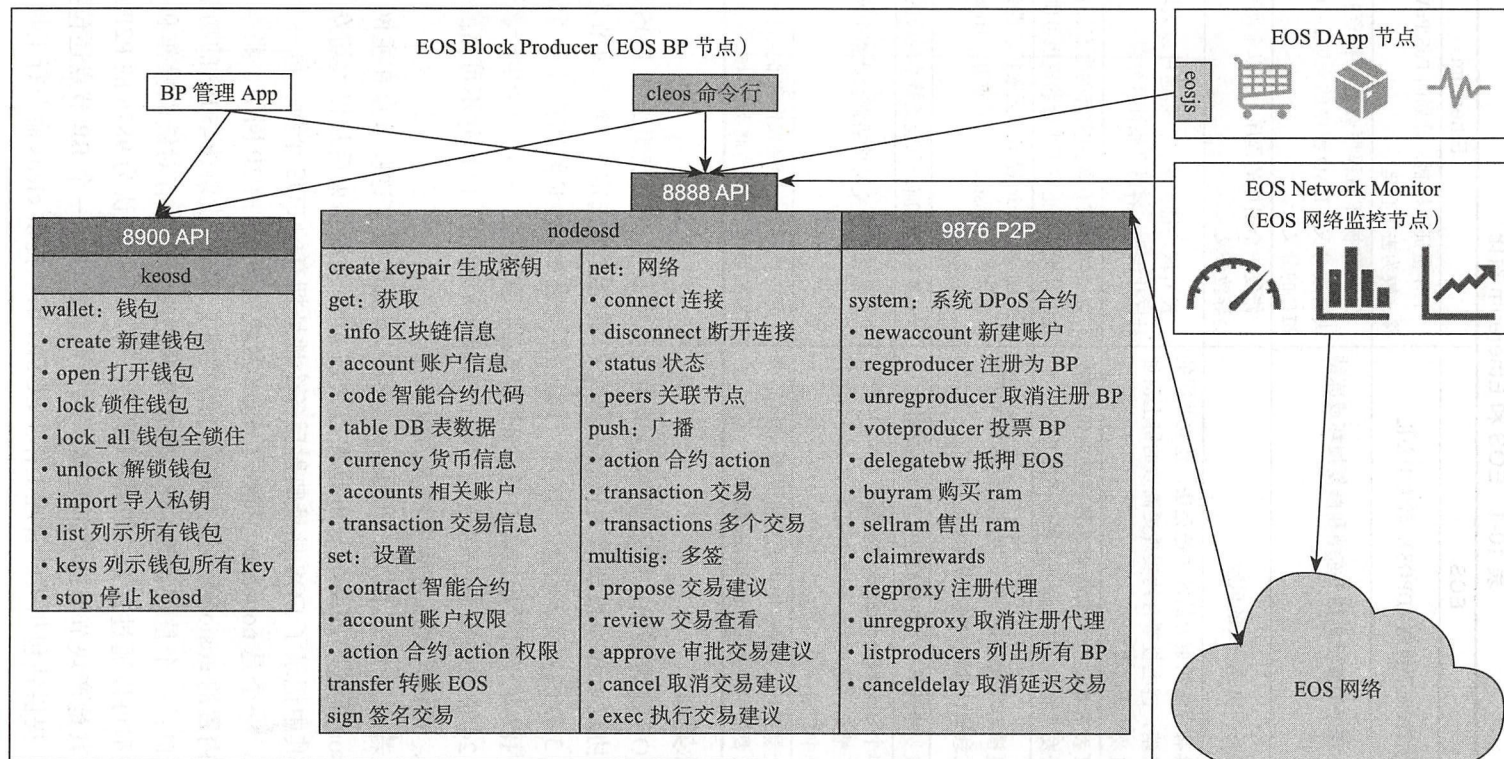


图 10-3 EOS 部署架构



nodeosd 的 8888 API 端口和 keosd 的 8900 API 端口, BP 可以实现对区块链和钱包的管理功能。BP 也可以开发出基于这两个端口的管理界面 App。对于分布式应用节点, 通过连接 nodeosd 的 8888 API 端口实现合约的部署和合约的应用操作, 目前, EOSIO 提供一个叫 eosjs 的 node.js 项目作为开发工具包, 用户的 DApp 可以进行集成。EOSIO 或者其他管理平台, 可以通过连接 nodeosd 的 8888 API 端口获得每个 BP 节点的状态, 如当前块的高度; 同时基于 eosio 合约可以获得当前 BP 产块者、每个 BP 所产区块个数等信息; 基于 API, 在 EOS 的生态里, 可以存在 EOS 的网络监控节点, 提供整个网络的健康视图。

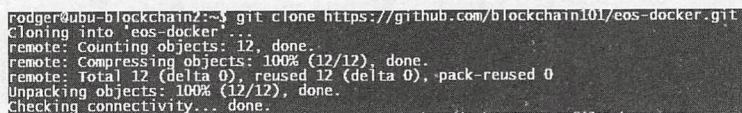
### 3. cleos 命令行举例

我们以一个私链为例, 说明在 EOS 链上主要的命令操作, 如生成账户、部署合约。

#### 1) 从 GitHub 获取 eos-docker 资源文件:

```
$ git clone https://github.com/blockchain101/eos-docker.git
```

执行结果如图 10-4 所示。



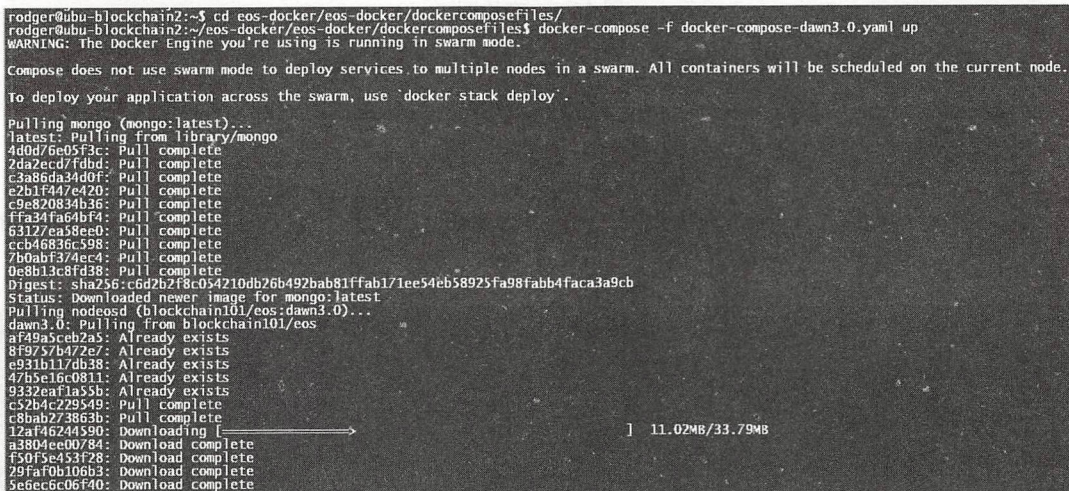
```
rodger@ubu-blackchain2:~$ git clone https://github.com/blockchain101/eos-docker.git
Cloning into 'eos-docker'...
remote: Counting objects: 12, done.
remote: Compressing objects: 100% (12/12), done.
remote: Total 12 (delta 0), reused 12 (delta 0), pack-reused 0
Unpacking objects: 100% (12/12), done.
Checking connectivity... done.
```

图 10-4 获取 eos-docker 资源文件

#### 2) 执行 dockercompose 文件, 创建一个私链:

```
$ cd eos-docker/eos-docker/dockercomposefiles
$ docker-compose -f docker-compose-dawn3.0.yaml up
```

第一次运行时下载相关的 images, 如图 10-5 所示。



```
rodger@ubu-blackchain2:~$ cd eos-docker/eos-docker/dockercomposefiles/
rodger@ubu-blackchain2:~/eos-docker/eos-docker/dockercomposefiles$ docker-compose -f docker-compose-dawn3.0.yaml up
WARNING: The Docker Engine you're using is running in swarm mode.

Compose does not use swarm mode to deploy services to multiple nodes in a swarm. All containers will be scheduled on the current node.
To deploy your application across the swarm, use 'docker stack deploy'.

Pulling mongo (mongo:latest)...
latest: Pulling from library/mongo
4d0d76e05f3c: Pull complete
2da2ecd7fdbd: Pull complete
c3a86da24d0f: Pull complete
e2b1f447e420: Pull complete
c9e820834b36: Pull complete
ffa34fa64bf4: Pull complete
63127ea38ee0: Pull complete
ccb46836c598: Pull complete
7b0ab1374ecd: Pull complete
0e8b13c8fd38: Pull complete
Digest: sha256:c6d2b2f8c054210db26b492bab81ffab171ee54eb58925fa98fab4faca3a9cb
Status: Downloaded newer image for mongo:latest
Pulling nodeosd (blockchain101/eos:dawn3.0)...
dawn3.0: Pulling from blockchain101/eos
a748a3ceb2a5: Already exists
8f9757b472e7: Already exists
e931b117db38: Already exists
47b5e16c0811: Already exists
9332eaf1a55b: Already exists
c52b4228549: Pull complete
c8b0b273863b: Pull complete
12af46244590: Downloading [=====>] 11.02MB/33.79MB
a3804ee00784: Download complete
f50f5e453f28: Download complete
29faf0b106b3: Download complete
5e6ec6c06f40: Download complete
```

图 10-5 下载相关的 images



等所有的 image 下载完毕后, 会启动相关的容器 nodeosd 和 keosd, 并且由单节点 eosio 账户执行产块, 可以看到如图 10-6 所示的界面。

```
nodeosd: *****
nodeosd: *
nodeosd: *      NEW CHAIN      *
nodeosd: *  -- Welcome to EOSIO --  *
nodeosd: *
nodeosd: *****
nodeosd: Your genesis seems to have an old timestamp
nodeosd: Please consider using the --genesis-timestamp option to give your genesis a recent timestamp
nodeosd:
287/3127ms thread-0 producer plugin.cpp:176 plugin_startup producer plugin: plugin_startup() end
nodeosd: 287/3127ms thread-0 http plugin.cpp:285 plugin_startup Start listening for http requests
nodeosd: 287/3127ms thread-0 chain api plugin.cpp:62 plugin_startup starting chain api plugin
nodeosd: 287/3127ms thread-0 http plugin.cpp:325 add handler add api url: /v1/chain/abi_bin_to_json
nodeosd: 287/3127ms thread-0 http plugin.cpp:325 add handler add api url: /v1/chain/abi_json_to_bin
nodeosd: 287/3127ms thread-0 http plugin.cpp:325 add handler add api url: /v1/chain/get_account
nodeosd: 287/3127ms thread-0 http plugin.cpp:325 add handler add api url: /v1/chain/get_block
nodeosd: 287/3127ms thread-0 http plugin.cpp:325 add handler add api url: /v1/chain/get_code
nodeosd: 287/3127ms thread-0 http plugin.cpp:325 add handler add api url: /v1/chain/get_currency_balance
nodeosd: 287/3127ms thread-0 http plugin.cpp:325 add handler add api url: /v1/chain/get_currency_stats
nodeosd: 287/3127ms thread-0 http plugin.cpp:325 add handler add api url: /v1/chain/get_info
nodeosd: 287/3127ms thread-0 http plugin.cpp:325 add handler add api url: /v1/chain/get_required_keys
nodeosd: 287/3127ms thread-0 http plugin.cpp:325 add handler add api url: /v1/chain/get_table_rows
nodeosd: 287/3127ms thread-0 http plugin.cpp:325 add handler add api url: /v1/chain/push_block
nodeosd: 287/3127ms thread-0 http plugin.cpp:325 add handler add api url: /v1/chain/push_transaction
nodeosd: 287/3127ms thread-0 http plugin.cpp:325 add handler add api url: /v1/chain/push_transactions
nodeosd: 287/3127ms thread-0 wallet api plugin.cpp:70 plugin_startup starting wallet api plugin
nodeosd: 287/3127ms thread-0 http plugin.cpp:325 add handler add api url: /v1/wallet/create
nodeosd: 287/3127ms thread-0 http plugin.cpp:325 add handler add api url: /v1/wallet/get_public_keys
nodeosd: 287/3127ms thread-0 http plugin.cpp:325 add handler add api url: /v1/wallet/import_key
nodeosd: 287/3127ms thread-0 http plugin.cpp:325 add handler add api url: /v1/wallet/list_keys
nodeosd: 287/3127ms thread-0 http plugin.cpp:325 add handler add api url: /v1/wallet/list_wallets
nodeosd: 287/3127ms thread-0 http plugin.cpp:325 add handler add api url: /v1/wallet/lock
nodeosd: 287/3127ms thread-0 http plugin.cpp:325 add handler add api url: /v1/wallet/lock_all
nodeosd: 287/3127ms thread-0 http plugin.cpp:325 add handler add api url: /v1/wallet/open
nodeosd: 287/3127ms thread-0 http plugin.cpp:325 add handler add api url: /v1/wallet/set_timeout
nodeosd: 287/3127ms thread-0 http plugin.cpp:325 add handler add api url: /v1/wallet/sign_transaction
nodeosd: 287/3127ms thread-0 http plugin.cpp:325 add handler add api url: /v1/wallet/unlock
nodeosd: 287/3127ms thread-0 account history api plugin.cpp:45 plugin_startup starting account history api plugin
nodeosd: 287/3127ms thread-0 http plugin.cpp:325 add handler add api url: /v1/account/history/get_controlled_accounts
nodeosd: 287/3127ms thread-0 http plugin.cpp:325 add handler add api url: /v1/account/history/get_key_accounts
nodeosd: 287/3127ms thread-0 http plugin.cpp:325 add handler add api url: /v1/account/history/get_transaction
nodeosd: 287/3127ms thread-0 http plugin.cpp:325 add handler add api url: /v1/account/history/get_transactions
nodeosd: 287/3127ms thread-0 net plugin.cpp:2902 plugin_startup Starting listener, max clients is 25
nodeosd: 287/3127ms thread-0 producer plugin.cpp:239 block_production_loop eosio generated block 115219... #1 @ 2018-05-20T14:47:53.500 with 0 trxs, lib: 0
nodeosd: keosd: 287/3127ms thread-0 wallet plugin.cpp:41 plugin_initialize configured http to listen on 127.0.0.1:8888
nodeosd: keosd: 287/3127ms thread-0 http plugin.cpp:285 plugin_startup Start listening for http requests
nodeosd: keosd: 287/3127ms thread-0 wallet api plugin.cpp:70 plugin_startup starting wallet api plugin
nodeosd: keosd: 287/3127ms thread-0 http plugin.cpp:325 add handler add api url: /v1/wallet/create
nodeosd: keosd: 287/3127ms thread-0 http plugin.cpp:325 add handler add api url: /v1/wallet/get_public_keys
nodeosd: keosd: 287/3127ms thread-0 http plugin.cpp:325 add handler add api url: /v1/wallet/import_key
nodeosd: keosd: 287/3127ms thread-0 http plugin.cpp:325 add handler add api url: /v1/wallet/list_keys
nodeosd: keosd: 287/3127ms thread-0 http plugin.cpp:325 add handler add api url: /v1/wallet/list_wallets
nodeosd: keosd: 287/3127ms thread-0 http plugin.cpp:325 add handler add api url: /v1/wallet/lock
nodeosd: keosd: 287/3127ms thread-0 http plugin.cpp:325 add handler add api url: /v1/wallet/lock_all
nodeosd: keosd: 287/3127ms thread-0 http plugin.cpp:325 add handler add api url: /v1/wallet/open
nodeosd: keosd: 287/3127ms thread-0 http plugin.cpp:325 add handler add api url: /v1/wallet/set_timeout
nodeosd: keosd: 287/3127ms thread-0 http plugin.cpp:325 add handler add api url: /v1/wallet/sign_transaction
nodeosd: keosd: 287/3127ms thread-0 http plugin.cpp:325 add handler add api url: /v1/wallet/unlock
nodeosd: keosd: 287/4001ms thread-0 producer plugin.cpp:239 block_production_loop eosio generated block 336069... #7 @ 2018-05-20T14:47:54.000 with 0 trxs, lib: 1
nodeosd: nodeosd: 287/4501ms thread-0 producer plugin.cpp:239 block_production_loop eosio generated block 5be38e92... #1 @ 2018-05-20T14:47:54.500 with 0 trxs, lib: 2
nodeosd: nodeosd: 287/5001ms thread-0 producer plugin.cpp:239 block_production_loop eosio generated block c6b0d8... #1 @ 2018-05-20T14:47:55.000 with 0 trxs, lib: 3
nodeosd: nodeosd: 287/5054ms thread-0 producer plugin.cpp:239 block_production_loop eosio generated block 2162a3b2... #5 @ 2018-05-20T14:47:56.000 with 0 trxs, lib: 4
nodeosd: nodeosd: 287/5054ms thread-0 producer plugin.cpp:239 block_production_loop eosio generated block 13463b2a... #6 @ 2018-05-20T14:47:56.500 with 0 trxs, lib: 5
nodeosd: nodeosd: 287/5054ms thread-0 producer plugin.cpp:239 block_production_loop eosio generated block f0c1c670... #7 @ 2018-05-20T14:47:57.000 with 0 trxs, lib: 6
nodeosd: nodeosd: 287/5054ms thread-0 producer plugin.cpp:239 block_production_loop eosio generated block 22d948af... #8 @ 2018-05-20T14:47:57.500 with 0 trxs, lib: 7
nodeosd: nodeosd: 287/5054ms thread-0 producer plugin.cpp:239 block_production_loop eosio generated block 8179c2b... #9 @ 2018-05-20T14:47:58.000 with 0 trxs, lib: 8
nodeosd: nodeosd: 287/5054ms thread-0 producer plugin.cpp:239 block_production_loop eosio generated block 304aa8b9... #10 @ 2018-05-20T14:47:58.500 with 0 trxs, lib: 9
nodeosd: nodeosd: 287/5054ms thread-0 producer plugin.cpp:239 block_production_loop eosio generated block 453b07f... #11 @ 2018-05-20T14:47:59.000 with 0 trxs, lib: 10
nodeosd: nodeosd: 287/5054ms thread-0 producer plugin.cpp:239 block_production_loop eosio generated block 3d1a0853... #12 @ 2018-05-20T14:47:59.500 with 0 trxs, lib: 11
nodeosd: nodeosd: 287/5054ms thread-0 producer plugin.cpp:239 block_production_loop eosio generated block 7933d0b... #13 @ 2018-05-20T14:48:00.000 with 0 trxs, lib: 12
nodeosd: nodeosd: 287/5054ms thread-0 producer plugin.cpp:239 block_production_loop eosio generated block c141692c... #14 @ 2018-05-20T14:48:00.500 with 0 trxs, lib: 13
nodeosd: nodeosd: 287/5054ms thread-0 producer plugin.cpp:239 block_production_loop eosio generated block 1a6b5b2d... #15 @ 2018-05-20T14:48:01.000 with 0 trxs, lib: 14
nodeosd: nodeosd: 287/5054ms thread-0 producer plugin.cpp:239 block_production_loop eosio generated block b2629600... #16 @ 2018-05-20T14:48:01.500 with 0 trxs, lib: 15
nodeosd: nodeosd: 287/5054ms thread-0 producer plugin.cpp:239 block_production_loop eosio generated block 5d814b5... #17 @ 2018-05-20T14:48:02.000 with 0 trxs, lib: 16
nodeosd: nodeosd: 287/5054ms thread-0 producer plugin.cpp:239 block_production_loop eosio generated block 3d814b5... #18 @ 2018-05-20T14:48:02.500 with 0 trxs, lib: 17
nodeosd: nodeosd: 287/5054ms thread-0 producer plugin.cpp:239 block_production_loop eosio generated block 9ab7b2b... #19 @ 2018-05-20T14:48:03.000 with 0 trxs, lib: 18
```

图 10-6 产块图

### 3) 创建一个 EOS 公私钥地址:

```
$ alias cleos='docker exec keosd /opt/eosio/bin/cleos -u http://nodeosd:8888'
$ cleos get info # 获取当前区块信息
$ cleos create key # 生成一个新的 EOS 公私钥地址
```

我们为 keosd 容器中运行命令指定一个 alias cleos, 这样直接使用 cleos 就可以执行命令, 如图 10-7 所示。

```
rodger@ubu-blockchain2:~/eos-docker/eos-docker/dockercomposefiles$ alias cleos='docker exec keosd /opt/eosio/bin/cleos -u http://nodeosd:8888'
rodger@ubu-blockchain2:~/eos-docker/eos-docker/dockercomposefiles$ cd
rodger@ubu-blockchain2:~$ alias cleos='docker exec keosd /opt/eosio/bin/cleos -u http://nodeosd:8888'
rodger@ubu-blockchain2:~$ cleos get info
{
  "server_version": "2ae61517",
  "head_block_num": 1066,
  "last_irreversible_block_num": 1065,
  "head_block_time": "2018-05-20T15:04:09",
  "head_block_producer": "eosio"
}
rodger@ubu-blockchain2:~$ cleos create key
Private key: 5KMH74FuWUj2bmL1fFCVggemIE7YAHpVpZonssVWmU586fb1LqS
Public key: E0S64f39KJvzJgrpxeoxk1zkWlhUpmcB28rSzdZQRtCbdcJX
```

图 10-7 使用 cleos 执行命令



4) 创建一个钱包。默认的钱包中会包括 eosio 账户私钥，现在将刚刚创建的私钥导入钱包：

```
$ cleos wallet create # 创建一个钱包，默认为 default.wallet
$ cleos wallet list # 列示所有钱包，此时只有 default.wallet
$ cleos wallet keys # 列出所有钱包中的 keypairs
$ cleos wallet import '私钥' # 导入私钥到默认钱包 default.wallet
```

我们先创建一个钱包，然后把新创建的私钥导入到钱包中去，再次查看 keys 时，会发

现新的公私钥对已经在钱包中了，如图 10-8 所示。

```
rodger@ubu-blockchain2:~$ cleos wallet create
Creating wallet: default
Save password to use in the future to unlock this wallet.
Without password imported keys will not be retrievable:
"Pw5KTwccUwQ3zXkvH46n91e8SpvD3dFiIXLe1Dw4gs/zLtsFh1MP"
rodger@ubu-blockchain2:~$ cleos wallet list
Wallets:
[
  "default *"
]
rodger@ubu-blockchain2:~$ cleos wallet keys
[[
  "EOS6MRyAjQq8ud7hVNYcfnVPJqcVpscN5So8BhtHuGYqET5GDW5CV",
  "5KQwrPbwdL6PhXujxW37FSSQZ1J1wsST4cqQzDeyXtP79zkvFD3"
]]
rodger@ubu-blockchain2:~$ cleos wallet import 5Kmh74FWUj2bmLiffCVgqemTE7YAhPvpZonssvWmmU586fbBilQs
imported private key for: EOS64F397k3vyZ3grxpEoxkizkw1hUgwCb28rSzdZqRRtcb8dcjX
rodger@ubu-blockchain2:~$ cleos wallet keys
[[
  "EOS64F397k3vyZ3grxpEoxkizkw1hUgwCb28rSzdZqRRtcb8dcjX",
  "5Kmh74FWUj2bmLiffCVgqemTE7YAhPvpZonssvWmmU586fbBilQs",
  "EOS6MRyAjQq8ud7hVNYcfnVPJqcVpscN5So8BhtHuGYqET5GDW5CV",
  "5KQwrPbwdL6PhXujxW37FSSQZ1J1wsST4cqQzDeyXtP79zkvFD3"
]]
```

图 10-8 导入新创建的私钥

5) 通过 eosio 账户创建一个合约账户 eosio.token (后面会使用 eosio.token 账户创建合约)：

```
$ cleos create account eosio eosio.token 'owner 公钥' 'active 公钥'
```

这里我们设置 owner 公钥和 active 公钥刚刚新产生的相同的公钥，一般设置成 2 个公钥。为安全起见，owner 私钥离线保存，active 私钥才导入钱包进行交易签名。使用 owner 私钥可以恢复账户，如图 10-9 所示。

```
rodger@ubu-blockchain2:~$ cleos create account eosio eosio.token EOS64F397k3vyZ3grxpEoxkizkw1hUgwCb28rSzdZqRRtcb8dcjX
executed transaction: d57584e840812ec0b8393caa4c458057162b3c9f825806cd32928c7684de399c... 332 bytes 102400 cycles
eosio -- eosio -- newaccount { "creator": "eosio", "name": "eosio.token", "owner": "[ "key": "EOS64F397k3vyZ3grxpEo...
```

图 10-9 设置 2 个公钥

6) 使用系统默认合约模板，创建智能合约 eos.token：

```
$ cleos set contract eosio.token contracts/eosio.token -p eosio.token
```

我们使用系统合约 eosio.token 模板创建一个 token 合约，-p 指定使用 eosio.token 账户的私钥进行交易签名，执行结果如图 10-10 所示。

7) 创建 token，指定 token 发行人、发行规模：

```
$ cleos push action eosio.token create '[ "eosio", "1000000000.0000 ZPT", 0, 0, 0]' -p eosio.token
```







```

rodger@ubu-blockchain2:~$ cleos push action eosio.token create '[ "eosio", "10000000.0000 EOS", "0, 0, 0" ]' -p eosio.token
executed transaction: 454cb206a9176904b26ca904ccb36d4b5976d40305b165944bdcc5a93f9454bd 248 bytes 104448 cycles
# eosio.token <= eosio.token::create [{"issuer": "eosio", "maximum_supply": "10000000.0000 EOS", "can_freeze": "0", "can_recall": "0", "can_whitelist...
rodger@ubu-blockchain2:~$ cleos push action eosio.token issue '[ "eosuser", "999999000.0000 EOS", "issue to eosuser 999999000 EOS" ]' -p eosio
executed transaction: df572f8c375163bb64dda512839341ef748c3497e4e0123600ee437e9f1cd8 280 bytes 139264 cycles
# eosio.token <= eosio.token::issue [{"to": "eosuser", "quantity": "999999000.0000 EOS", "memo": "issue to eosuser 999999000 EOS"}]
>> issue
# eosio.token <= eosio.token::transfer [{"from": "eosio", "to": "eosuser", "quantity": "999999000.0000 EOS", "memo": "issue to eosuser 999999000 EO...
>> transfer
# eosio <= eosio.token::transfer [{"from": "eosio", "to": "eosuser", "quantity": "999999000.0000 EOS", "memo": "issue to eosuser 999999000 EO...
# eosuser <= eosio.token::transfer [{"from": "eosio", "to": "eosuser", "quantity": "999999000.0000 EOS", "memo": "issue to eosuser 999999000 EO...
rodger@ubu-blockchain2:~$ cleos push action eosio.token issue '[ "eosio", "1000.0000 EOS", "issue to eosio the rest EOS" ]' -p eosio
executed transaction: 26344eab1d6013144d5a91534272e8f3d9d41de4aa54089f9a5820dacaac1c71e 280 bytes 112640 cycles
# eosio.token <= eosio.token::issue [{"to": "eosio", "quantity": "1000.0000 EOS", "memo": "issue to eosio the rest EOS"}]
>> issue
rodger@ubu-blockchain2:~$ cleos get currency balance eosio.token eosuser
1000.0000 EOS
rodger@ubu-blockchain2:~$ cleos get currency balance eosio.token eosio
999999000.0000 ZPT

```

图 10-14 EOS 资产创建和发行的过程

### 10) 通过 eosio.token 的 tranfer action 执行资产的转移:

```
$ cleos push action eosio.token transfer '[ "eosio", "eosuser", "500.0000 EOS",
"transfer to eosuser 500 EOS" ]' -p eosio
```

我们转移 500 EOS 给 eosuser, transfer 执行后, 查看双方的余额情况, 如图 10-15 所示。

```

rodger@ubu-blockchain2:~$ cleos push action eosio.token transfer '[ "eosio", "eosuser", "500.0000 EOS", "transfer to eosuser 500 EOS" ]' -p eos
executed transaction: 5e4641d3042a041e47e8dc160d7819170facf8fc3984586857a9e4b2b5c91450 288 bytes 116736 cycles
# eosio.token <= eosio.token::transfer [{"from": "eosio", "to": "eosuser", "quantity": "500.0000 EOS", "memo": "transfer to eosuser 500 EOS"}]
>> transfer
# eosio <= eosio.token::transfer [{"from": "eosio", "to": "eosuser", "quantity": "500.0000 EOS", "memo": "transfer to eosuser 500 EOS"}]
# eosuser <= eosio.token::transfer [{"from": "eosio", "to": "eosuser", "quantity": "500.0000 EOS", "memo": "transfer to eosuser 500 EOS"}]
rodger@ubu-blockchain2:~$ cleos get currency balance eosio.token eosio
500.0000 EOS
rodger@ubu-blockchain2:~$ cleos get currency balance eosio.token eosuser
999999000.0000 ZPT
rodger@ubu-blockchain2:~$ cleos get currency balance eosio.token eosuser
999999500.0000 EOS
1000.0000 ZPT

```

图 10-15 查看双方余额

EOS 其他合约的部署和执行同上述 eosio.token 的部署和执行类似, 这里不再赘述。除了 EOS 以外, 还有另一个区块链操作系统项目叫 Elastos, 有兴趣的读者也可以自行关注学习。

### 10.3.3 Cardano: 具有隐私和法规的区块链

Cardano (Ada), 中文翻译为卡尔达诺链 (艾达币)。Cardano 是应用平台, ADA 是它的数字货币。Cardano 是世界上第一个以研究为主导, 从科学哲学开发出来的区块链项目, 也是第一个采用同行评审技术的区块链项目, 可用于发送和接收数字资金, 支持各种去中心化应用和智能合约。Cardano 通过 Ouroboros 这种可验证的安全证明算法及共识机制来提高交易吞吐量, 通过优化网络带宽以及管理数据扩展, 来提高区块链的可扩展性。Gardano 的相关资料参考网址如下。

- 官网: <https://www.cardano.org>;
- 源码: <https://github.com/input-output-hk/cardano-sl>;
- 白皮书: <https://www.cardano.org/en/academic-papers>。

由于 ADA 幕后技术团队 IOHK 的 CEO Charles Hoskinson 是以太坊的合伙创始人, 同时也是前以太坊 CEO 和 Bitshare 创始人之一, 所以 Cardano 深入研究了以太坊的不足, 且



做了很多层面的改进甚至革命性的创新。坊间戏称 Cardano 为“日本的以太坊”，因为最初启动团队以及现在的社区运营方是日本项目商 Emurgo，且 2017 年 9 月底 ICO 时 90% 以上的 Cardano 持有者都是做过 KYC 身份认证的日本投资者。Cardano 目前虽然只开源了结算层（settlement layer）的实现源码，但是其 github 上有 13 000 多个 commits，项目非常活跃。Cardano 经常被人拿来和 EOS 对比，因为 ADA 的 Charles Hoskinson 和 EOS 的 Dan Larimer 之前一起合作过 Bitshares 项目。

### （1）共识算法

Cardano 采用 Proof-of-Stake 权益证明作为其共识机制。权益证明使用币的年龄、币的数量和有关持有人钱包的其他因素，而不是使用工作量证明 PoW。权益证明是关于新块和交易的验证，而不是开采新块。为了防止错误的交易和区块被发布，需要某种形式的证明。这是加密货币与法定货币不同的一个决定性特征。

### （2）含尾蛇协议

Cardano 引进了基于 PoS 权益证明的共识算法协议——“含尾蛇协议”（Ouroboros Protocol）。这是一个安全的权益证明体系，旨在防止不良行为者利用可能的随机性质量不高的问题。同时从一维运作去到三维运作，多个链条同时进行，随着链条节点的增加，Cardano 每秒可以处理非常庞大的交易量。Ouroboros 为领导者选举过程产生随机性的创新是通过安全的多方实施代币协议来实现的。Cardano 正在用形式化的建模语言 Psi Calculus 来建模 Ouroboros 和其他协议功能，这样在实际实现协议之前，就能对它们进行测试。

### （3）模块化（Modularity）的双层协议机制

为了解决可扩展性（Scalability）、互操作性（Interoperability）和可持续性（Sustainability），Cardano 设计了模块化的双层协议机制：Cardano Settlement Layer（CSL）结算层负责存放账户信息；Cardano Computation Layer（CCL）计算层负责处理结算层中的信息以及执行智能合约。

### （4）递归互联网架构

Cardano 使用 Recursive Inter Network Architecture（RINA）递归互联网架构把整个网络切分成互相沟通的多个子网络，一个子网络里有很多节点，每个节点只负责存储被分配到的子网络的数据。

### （5）同行评审（peer review）

Cardano 采用同行评审流程，并将密码学、数学和信息安全专家的反馈融合在一起。这个过程能够使加密货币有能力更好地处理未来的需求。

### （6）编程语言和钱包

Cardano 支持 Haskell 编程语言。Haskell 是一种标准化的，通用的纯函数编程语言。Haskell 具有“证明即程序、命题为类型”的特征。Haskell 是强类型语言，对智能合约来说，Haskell 具有严谨性和安全性。Daedalus 钱包作为 ADA 目前唯一的官方钱包，支持多个其他币种储存功能。





### (7) 非交互式工作量证明的证明 (NIPoPoW)

非交互式工作量证明的证明 Non-interactive Proofs of Proof of Work (NIPoPoW), 是 Cardano 引入的一种可证实的安全侧链技术, 可以修复所有已知的攻击途径, 为提高侧链的安全性开辟新的道路。此项技术目前还没有实施。

### (8) Ouroboros Praos

Praos 是升级 Ouroboros 的一项举措。Praos 是半同步的, 不需要繁重的 MPC 协议来实现随机性, 并鼓励对输入支持者和插槽领导者采用抗量子签名方案。Praos 允许更短的出块时间。

### (9) Ouroboros Hydra

Hydra 是 Ouroboros 的另一个升级, 引入了分片 Sharding, 允许同时运行多个 Ouroboros 时期, 这显著增加了 Cardano 的吞吐量。分片最大的困难之一是分片间通信。从目前已有的材料看, Hydra 很可能是一个 DAG 结构的实现, 让我们拭目以待。

### (10) Cardano 主要设计原则

- ❑ 更广泛的金融普惠;
- ❑ 符合政府 / 监管机构的期望;
- ❑ 探索“商业的社会因素”。

## 10.3.4 基于区块链的支付协议: Ripple 与 Stellar

### 1. Ripple

Ripple 是一种关于支付的通用协议, 它可以让独立系统像邮件系统那样互联起来, 是一个可以让全世界服务器互相之间进行点对点金融交易的开源软件, 是世界上第一个开放的支付网络。通过它可以转账任意一种货币, 包括法币和加密货币等。交易确认在几秒内就能快速完成, 交易费用几乎为零, 没有所谓的跨行异地以及跨国支付费用。Ripple 相关资料的网址如下。

- ❑ 官网: <https://ripple.com>
- ❑ 源码: <https://github.com/ripple>
- ❑ 白皮书: [https://ripple.com/files/ripple\\_consensus\\_whitepaper.pdf](https://ripple.com/files/ripple_consensus_whitepaper.pdf)

Ripple 官方称 Ripple 是专为企业使用而设计的, 通过 RippleNet 连接银行、支付服务供应商和数字货币交易平台, 为全球超过 155 亿美元的跨境支付市场提供零阻力体验。为银行和支付服务提供商提供了一个获取国际支付流动性的可靠和应需的途径。你可以把 Ripple 看作一个去中心化或弱中心化的分布式清算系统。Ripple 网络是基于共享的公开数据库, 记录账号和结余的总账, 用来实现支付交易和自动清算。任何人都可以查看这些总账和交易活动记录。因为 Ripple 是分布式网络, 没有单点故障, 所以可靠性也相对更好些, 另外因为网络是开源的, 所以也相对更加安全。随着 Ripple 的发展和被采用, 很可能补充、



甚至威胁到目前跨境汇款需要采用的国际间清算机构 SWIFT 的地位。SWIFT 组织会对每一笔资金结算进行收费,目前电汇一次的手续费为 15 美金。资金到账的时间也比较久,往往需要 24 小时以上。传统系统需要 3~5 天的清结算过程,Ripple 只需要 4 秒。

### (1) Ripple 的三款产品

通过 Ripple 网络的交易分为 2 种:使用 Ripple 系统原生代币 XRP 进行的交易(xRapid)和使用法定货币进行的交易(xCurrent)。xCurrent 的交易模式是专为银行等机构设计的,在 AML/KYC 上做了很多设计,打消了银行最大的顾虑。xRapid 跨境转账系统运用加密货币 XRP 作为媒介,解决跨境转账的信用和担保问题,帮助用户最大程度地节省流动性成本。Ripple 网络上所传递的价值还是各种货币本身的价值,XRP 有点类似于以太坊网络中 Gas 的作用,对资源使用费用进行计费。xVia 系统是 Ripple 的第三款产品,主要用户是非银行金融机构和公司,主要解决跨网络和跨境转账标准不同的问题,其特点是简单易用。

### (2) 开源的 Interledger 协议(ILP)

不同于常规分布式账本项目将所有信息记录在一份账本里,并在整个网络中共识确认的方式,Ripple 网络,将各个账本实时连接在一起,并增加了账本间的互通性。这使其允许不同银行间的不同账本互联互通,并迅速地处理和确认账户信息,资产在不同账本中的流转。Ripple 在 2015 年提出了 Interledger 协议。它本身不是一个区块链,而是一个简称为 ILP 的一套传输资产的规则。也就是说,ILP 解决的是不同账本不能互联互通的问题。ILP 有 4 个基本组件,一个是发送者(Sender),第二个是接收者(Receiver),第三个是连接器(Connector),第四个是账本(Ledger)。发送者和接收者都比较容易理解。连接器是连接发送者和接收者以完成 ILP 支付。连接器也可以连接其他连接器,这样可以将更多的发送者和接收者连接起来。账本是用来记录各账户的资产余额。在现实世界里,账本可以是传统银行或第三方支付账本,也可以是区块链的账本。如果发送者和接收者在同一个账本上,那么他们之间就不需要有链接器而可以直接转账。

类似 Internet 的架构有 4 层(应用层、传输层、跨网络层、网络层),Interledger 架构上也同样有 4 层:应用层、传输层、跨账本层和账本层。简单来说,Interledger 是将互联网的概念应用在互联账本上,以达到不同区块链之间、区块链与传统账本之间,甚至传统账本与账本之间能互联互通的目的。Interledger 在金融行业能解决账本孤岛的问题,提供安全、高效的跨账本互联,具有广阔的使用场景。

### (3) 瑞波协议共识算法(RPCA)

Ripple 网络上的每笔交易发出时,先经过本地节点的验证签名和交易合法性(签名是用公钥验证交易的签名,合法性主要是验证这笔钱是否可以花费等),再提交网络参与共识。共识就是唯一一节点列表(Unique Node List,简称 UNL)中的信任节点参与投票的过程。瑞波协议共识算法(Ripple Protocol consensus Algorithm, RPCA)实现了一种较高性能,同时拥有较高拜占庭容错的算法,每隔几秒会生成一个区块,然后应用到所有节点以便维护整





个网络的有效性和一致性。

#### (4) 防止“双花”

前文讲过，同一笔钱花两次就叫“双花”。Ripple 每提交一次交易先本地验证，然后提交网络共识，共识需要 3 ~ 5 秒钟。在共识未成功之前，可以把这笔钱再提交一次交易，在本地由于前一次还没有网络共识通过，所以此次交易还是能验证通过，然后提交网络共识。为了防止“双花”，Ripple 的解决方法就是依据共识的时间戳、先后顺序。如对于上述实例，第一次交易共识通过，第二次交易的共识就通不过了。

## 2. Stellar

Stellar (恒星)，是完全去中心的点对点支付网络，允许任何人发送和交换任意货币，包括法币和加密货币，可以在 2 ~ 5 秒内连接世界上的 180 多种货币，费用低并且简单。它是 Jed McCaleb (电驴创始人、Ripple 技术创始人) 在 Ripple 代码基础上创建的，可以称之为 Ripple2.0。与 Ripple 主要面向银行和企业不同，Stellar 是专门设计为平民百姓服务的。恒星也有自己的加密货币——恒星币 (XLM/Lumens)，相关资料参考以下网址。

□ 官网: <https://www.stellar.org>

□ 源码: <https://github.com/stellar>

□ 白皮书: <https://www.stellar.org/papers/stellar-consensus-protocol.pdf>

Stellar 充分借鉴了 Ripple 的共识协议和账户模型。在加密货币的经济模型上略有不同。

#### (1) 恒星共识协议

恒星共识协议 SCP (Stellar Consensus Protocol) 是首个安全可靠的联邦拜占庭协议 (FBA- Federated Byzantine Agreement) 的实现。FBA 的主要特点是权力下放和容忍任意行为。FBA 带来了开放的成员名单以及对拜占庭协议的去中心化控制，任何人都可以加入其中。通过分布式的方式，FBA 使得法定人数或者节点足够的群体能够达成一致。每个节点决定信任对象，不同的节点也不需要依赖于信赖相同的参与者组合即可完成共识。恒星共识协议具有 4 个关键属性：分散控制、低延迟、灵活信任和渐近安全 (Asymptotic Security)。

#### (2) 账户模型

Stellar 和 Ripple 一样，采用账户 (Account) 模型组织链上信息，所有信息都关联到账户。此类账户模型的设计相对于比特币采用的 UTXO 模型来讲，更加符合大众的一般认知。

Stellar 账户采用基于 Curve25519 的 ECDH 密码学算法。Stellar 公私钥账户地址如图 10-16 所示，其中 account id 表示公钥，secret key 表示密钥，用于对事务 (Transaction) 进行签名验签操作。密钥串能生成公钥和私钥 (通过 secret key 可以生成 account id)。这个 secret key 务必不要让任何人知道，任何人只要知道这个 key 就可以把你账户里的资金全部转走。



```
Generate Stellar testnet keys

{
  account id: GAU7HVDG2CVJU63XPW4S3MO3MS4J7K4GW3QCC267YYE6J3VOLWKPENCQ,
  secret key: SCOEIDSBKQ5DLPILCRVPHF2GBW3MVLKYWI7P6DKXPLV43XFHMIXF32I
}
```

图 10-16 Stellar 的公私钥账户地址

Ripple 和 Stellar 的一些对比如表 10-2 所示。

表 10-2 Ripple 和 Stellar 的对比

| 对比项      | Ripple (XRP)    | Stellar (XLM) |
|----------|-----------------|---------------|
| 共识机制     | RPCA            | FBA 联邦拜占庭共识   |
| 治理机制     | 中心化             | 去中心化          |
| 交易清算     | 2 ~ 4 秒         | 2 ~ 4 秒       |
| 团队       | 银行家             | 企业家           |
| 主要合作方    | 各大银行：美国银行、沙特央行等 | IBM 等         |
| 加密货币通货膨胀 | 静态 / 通缩         | 每年通货膨胀 1%     |

10.3.5 侧链代表：RootStock、Polkadot 和 Cosmos

侧链的概念是最早由 Blockstream 公司于 2014 年提出，其宗旨是在不改变主链结构和参数的情况下通过侧链来扩展主链的功能。如果一个链 SC 能看到链 MC 的所有交易，则称链 SC 为链 MC 的侧链，链 MC 为链 SC 的主链。其中主链 MC 并不知道侧链 SC 的存在，侧链 SC 知道主链 MC 的存在。

假设区块链 MC 是主链，SC 是 MC 的侧链，MC 和 SC 都拥有区块头和区块体，区块头中包括了交易构成的 Merkle 树根哈希。可以将主链 MC 的某区块头和该区块中包括的某交易的 Merkle 证明，作为侧链 SC 的交易写入链 SC 的区块中，如果主链 MC 采用的 PoX（如 PoW、PoS）最终确认性共识算法，在等待主链 MC 的一定个数的置信区块后，侧链 SC 就可以基于这个交易包含的 Merkle 证明信息来证明在链 A 上对应的交易操作。基于 Merkle 证明的区块链验证逻辑可以由链协议本身或应用合约来实现。一旦证明了交易的存在性，即确认了 A 用户向锁定账户转移了一定量的主链资产后，侧链就可以向用户 A 的侧链账户生成一定量的侧链代币。A 用户可以在侧链上将侧链代币转移给 B 用户。B 用户如果要把侧链资产转回到主链上，可以向侧链合约发送解锁交易，侧链合约就将一定量的主链资产转移给了 B 用户，同时在侧链上销毁对应量的侧链资产。上述过程如图 10-17 所示。

基于侧链的项目中，比较典型的有 Polkadot、Cosmos 和 Rootstock。这里我们来看看





Rootstock (RSK), 第一个和比特币双向锚定的、开源的智能合约平台。RSK 相关资料参考以下网址。

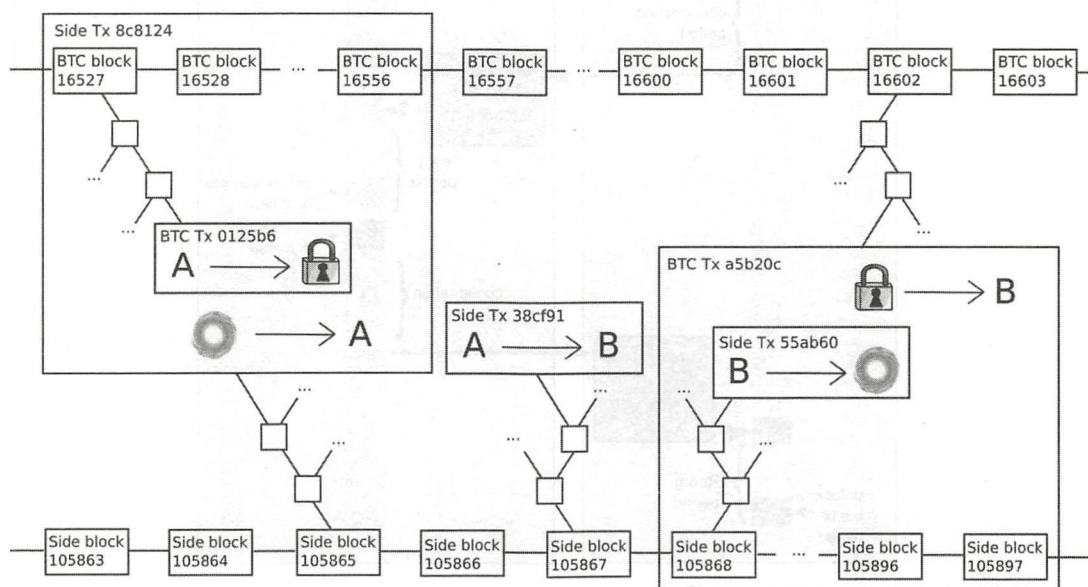


图 10-17 侧链模式资产转移流程

- 官网: <https://www.rsk.co>
- 源码: <https://github.com/rskmart>
- 白皮书: <https://www.rsk.co/#resources>

Rootstock 作为比特币的侧链存在, 不仅拥有比特币的所有功能, 而且其在几个方面增强了比特币: 图灵完备的虚拟机 (RVM) 智能合约; 10 秒内首次确认交易; PoW 联合挖矿; 低延迟嵌入对等网络; 侧链支持双向使用。Rootstock 的虚拟机加入了 EVM 的指令集, 因此兼容以太坊 EVM 上的智能合约。Rootstock 每 20 秒出块, 号称能支持每秒 300 笔交易, 并逐渐扩展到 2000TPS。虽然 Rootstock 有自己的区块链, 但是作为侧链, 它是和比特币区块链联合挖矿, 因此有着和比特币相同的安全性。

Rootstock 支持与比特币的双向挂钩。当资产从比特币主链上转到 Rootstock 的时候, 相应的比特币被锁定在主链, 而与之相当的 Rootstock 原生代币 RTC 会在 Rootstock 区块链上解锁。当要把 RTC 资产转到比特币主链时, 对应的 RTC 会在 Rootstock 上锁定, 而原先在比特币主链上锁定的相对应的 BTC 会被解锁, 如图 10-18 所示。

Rootstock 的这种侧链技术, 既能在不改变比特币主链架构的情况下, 扩展支持图灵完备的智能合约, 又能前向兼容以太坊智能合约, 还可以支持资产在主链和侧链间的流转, 支持即时支付。



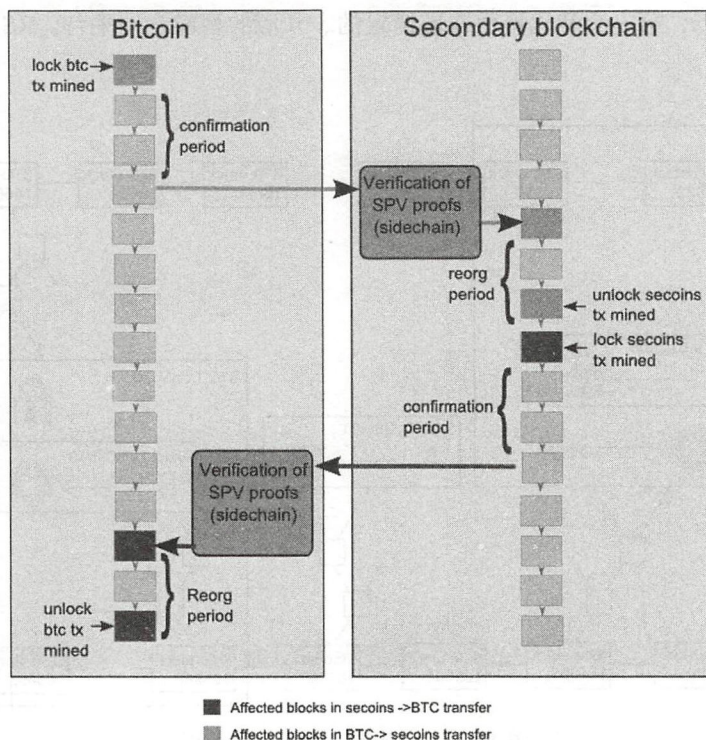


图 10-18 RootStock 侧链交易验证

### 10.3.6 分片扩容：Zilliqa 叫板 Visa

如果说侧链是通过“外部嫁接”到主链，那分片就是将主链进行“内部分割”。分片 (Sharding) 其实是一种传统数据库的技术，它将大数据库分成更小、更快、更容易管理的很多数据碎片，分而治之。此项技术应用到区块链中，会大幅提高区块链的扩展性。将区块链网络划分成若干能够处理交易的较小组件式网络，以实现每秒处理更多笔的交易。以太坊的 Sharding 和 EOS 的 Region 都是分片技术的实际应用。我们来看一下 Zilliqa，这个与 Visa 叫板的项目，声称有望用“分片”解决区块链的根本问题。

Zilliqa 起源于新加坡，是世界上首个通过分片技术 (sharding) 来实现可扩展性、高吞吐量的公有区块链平台，支持每秒数千次的交易，有望在建立稳定可靠的区块链基础设施后挑战每秒 8000 笔交易量的 Visa 和 MasterCard。Zilliqa 经亚马逊内部测试网测试，每秒实现的交易量突破了 2488 笔，是比特币和以太坊每秒交易笔数总和的 100 多倍。这还是前期阶段，如果以太坊的挖掘者 (约 22 000 人) 在 Zilliqa 的链上工作，那么 Zilliqa 每秒可支持 15 000 笔交易，相当于 Visa (全球最大支付处理商) 每秒交易数量的 2 倍多。Zilliqa 属于通过分片技术在链上直接扩容，而且交易吞吐量随其网络规模增加可以线性增加。Zilliqa 相关资料参考网址如下。





- ❑ 官网: <https://www.zilliqa.com/>
- ❑ 源码: <https://github.com/Zilliqa/Zilliqa>
- ❑ 白皮书: <https://docs.zilliqa.com/whitepaper.pdf>

Zilliqa 的技术特性如下。

- ❑ Zilliqa 属于链上项目。与侧链和链下项目相比,其在安全性和疏散性能方面更具优势,真正实现了主链的可扩展性。与此同时,Zilliqa 也可与侧链和链下的任何项目融合。
- ❑ Zilliqa 在测试网络中的交易处理速度约为以太坊的 250 倍。它采用高效优化的 pBFT 共识,以及使用 PoW 防止 Sybil 攻击。Zilliqa 的智能合约也与以太坊的智能合约不同,可以高效并发执行,提供真正安全高效的交易处理。
- ❑ Zilliqa 采用分片技术,每个片内都有 600 ~ 800 个节点来进行运算,保证安全性。同时,Zilliqa 不依赖于中心化的节点来处理分片过程。
- ❑ Zilliqa 的交易吞吐量随其网络规模(大致)线性增加。
- ❑ 网络分片 Network Sharding。
- ❑ DS 块,Sharding 结构,Shard 微块和 Final 块的共识。
- ❑ EC-Schnorr 签名。

### 10.3.7 垮链技术: 价值互联网的纽带

#### 1. PalletOne

PalletOne 是为了解决当前区块链技术存在的可扩展性、互操作性、用户友好性、平台独立性问题的一种分布式跨链协议,其本身不是一条区块链,而是高阶分布式共享账本共治协议。它希望成为区块链世界的 IP 协议。其相关资料参考网址如下。

- ❑ 官网: <https://pallet.io>
- ❑ 源码: <https://github.com/PalletOne>
- ❑ 白皮书: <https://pallet.io/doc/whitepaper.pdf>

Pallet (Protocol for Abstract-Level Ledger Ecosystem) 是多利益关联方的共识协议,所有的链都可以是参与方,不需要“链与链”之间的锚定。在 PalletOne 中,共识机制采用了陪审团,智能合约只需要一组验证人进行验证和执行,这些验证人被称为“陪审员”,并由他们组成“陪审团”。通过陪审团共识协议,PalletOne 将智能合约同底层区块链完全解耦,实现跨链价值交换。Mediator (调停中介) 负责 PalletOne 网络的安全性,是 PalletOne 的核心构成部分。PalletOne VM 是智能合约编译和执行的核心工具,是 PalletOne 支持多平台和多语言的关键部分。为了提升智能合约对通证定义的安全性,PalletOne 通证抽象层定义了关于通证的定义集和操作集。PalletOne 的架构和各个组成部分如图 10-19 所示。

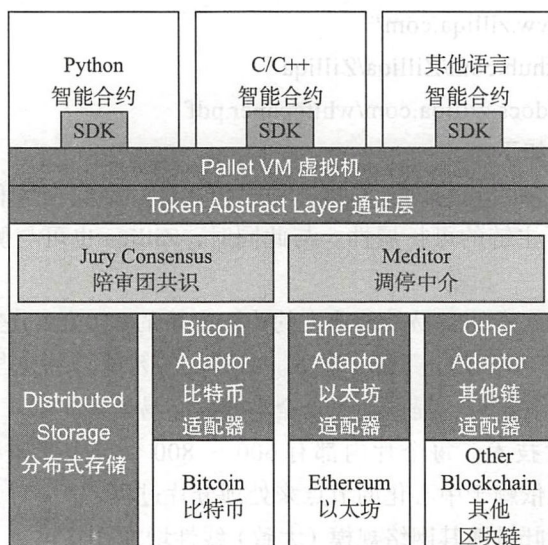


图 10-19 PalletOne 架构图

PalletOne 协议和体系不产生独立的链，而是采用陪审团共识机制随机选取陪审员，所有的陪审员节点再去对接各底层链实现存证，最大化利用现有的链和资源。Pallet 可以将应用挂在多个链上，避免网络拥塞，用户选择性大，使得各种 DApp 应用可以部署在不同的链上，形成良性竞争，更多用户可以参与。开发者在开发应用时可以根据业务需求更灵活地选择已有的链，或者方便地从已经部署的链上迁移到更好、更新的链上，从而使已有的应用生态突破原来部署区块链的自身架构局限、性能局限或者功能局限。由于 PalletOne 不是链，是通过特定的函数和库与底层链沟通，所以 PalletOne 的应用场景相对较广，并且实现难度上也处于中等水平。

### （1）PalletOne VM 虚拟机

作为智能合约编译和执行的核心工具，PalletOne VM 可以将主流编程语言（例如 C++、Python）编写的智能合约编译成为可以在不同平台上高效执行的字节码，使得智能合约不仅和底层区块链解耦，同时和智能合约语言、执行平台解耦。智能合约部署到 PalletOne 上后，将在 PalletOne VM 环境中运行。PalletOne VM 提供了一个对主机安全的沙盒环境，杜绝了恶意合约对主机或网络攻击的可能。

### （2）分布式存储

PalletOne 中将使用有向无环图（DAG）作为分布式存储。DAG 相比于传统的链式存储方式有许多优点。首先，在 DAG 中没有区块的概念，所有交易都独立封装在一个存储单元（Unit）中，单元之间通过引用建立连接关系。其次，使用 DAG 作为分布式存储，交易可以并行写入。再次，在 DAG 中，通过确定主链，使各个交易达到有序的状态，从而有效地解决了“双花”的问题。最后，DAG 中参加的节点越多，交易量越多，交易的确认速度越快，



因为交易之间是通过彼此引用的关系来进行确认的。在 PalletOne 分布式存储中, 需要存储的具体信息主要包括交易信息、合约 ID、合约代码、合约状态、合约对应的陪审员列表和陪审团在执行合约过程中处理的状态信息等。

### (3) 技术特性

- 多链、多任务、多语言、多平台;
- 开发安全性、合约执行安全性;
- 高性能、高通用;
- 采用 UTXO 模型提供类似比特币 P2PH、P2SH 等简单的支付体验。

基于多层次协议模式的跨链技术和区块链网络是非常值得期待的区块链 3.0 技术, 有兴趣的朋友还可关注如下协议和技术平台: Aion、Ontology、ArcBlock 等。

## 2. 其他平台

### (1) Aion

Aion, 世界首创的企业级多链协议, 专注构建一套为主流区块链应用提供更具广泛性和多样化解决方案的协议, 旨在支持协同性的完整、链上和链间的可扩展, 以及区块链的单独定制。在此设计中, Aion-1 作为通用区块链, 通过内在经济模式激励和保障网络整体运行。Aion 协议为链间交流定义了高性能的桥梁机制, 允许链间通过多个桥梁, 实现数据和价值的传递。Aion 相关资料参考以下网址。

- 官网: <https://aion.network/>;
- 源码: <https://github.com/aionnetwork>;
- 白皮书: <https://aion.network/whitepapers.html>。

### (2) Ontology

根据本体网络的白皮, Ontology 本体网络要做跨链协议, 是一个多链、多系统融合的协议网络, 来自不同的链和系统可以支持不同业务体系, 并通过本体网络的各类协议进行协作。本体网络本身有一条公有链, 提供分布式账本、智能合约、分布式实体管理认证协议、分布式数据交换协议等一系列的协议基础层与应用接口, 对于应用服务提供者来说, 无须具备底层的分布式系统开发能力就可以直接基于本体网络提供分布式服务。Ontology 相关资料参考以下网址。

- 官网: <https://ont.io/>
- 源码: <https://github.com/ontio>
- 白皮书: <https://ont.io/wp/Ontology-Introductory-White-Paper-EN.pdf>

### (3) ArcBlock

ArcBlock 是一个构建去中心化应用的生态系统。提供基础区块链服务组件, 将区块链能力集成到现有系统。它包含 Pub/Sub Gateway (发布订阅网关) 和 OpenChain Access Layer (通用层支持不同区块链), 并提供不同区块链的适配器 (Adapters), 也是一个比较典型的区块链中间件, 支持区块链跨链网络的组建。ArcBlock 相关资料参考以下网址。

❑ 官网: <https://www.arcblock.io>

❑ 源码: <https://github.com/ArcBlock>

❑ 白皮书: [https://www.arcblock.io/file/whitepaper/WhitePaperEnV2\\_en-US.pdf?v=4](https://www.arcblock.io/file/whitepaper/WhitePaperEnV2_en-US.pdf?v=4)

### 10.3.8 DAG: 区块链的革新

DAG 有向无环图 (Directed Acyclic Graph) 这个概念说新也不算新, 通常被用于数据处理、调度、寻找最优路径以及数据压缩等相关问题。数学专业或计算机专业的学生在运筹学或数据结构中应该学过 DAG 这个概念, 前几年业界也把 DAG 技术用在了图像处理 (如 3D 软件 Maya) 和大数据处理 (如 Spark 计算框架) 等领域。DAG 被用于区块链领域, 是区块链行业一次较大的创新, 代表产品有 IOTA、Byteball 和 TrustNote 等。其特点是不打包交易, 每笔交易作为一个单元, 不采用区块结构, 后来交易节点验证过往交易, 形成交易的有向无环图。后面也出现 Spectre Protocol 项目, 该协议将 DAG 与区块结合, 可以打包区块, 但区块之间是形成有向无环图, 而不是传统的一条链。DAG 的主要特点是高并发、确认快, 几乎在各个级别上都能表现出比区块链更优的特性, 业界认为 DAG 很有可能成为真正的区块链 3.0 技术标准。DAG 的难点在于共识机制以及主链选择以防“双花”, 也需要解决交易频率低时交易的验证问题。下面我们介绍主要的几个特色 DAG 项目: IOTA、Byteball 和 TrustNote。

#### 1. IOTA

IOTA 就如它的名字一样, 是专门给物联网 IoT 设备设计的事务处理和数据传输层。它提出了创新的分布式账本——Tangle (缠结), 摆脱了传统区块链设计的低效率问题, 引入在分布式点对点系统中达成共识的新方法。IOTA 的核心团队成员有 David Sonstebo、Sergey Ivancheglo、Serguei Popov 和 Dominik Schiener。据说这个团队之前参与开发了最早的一个基于 POS 的加密货币 NXT。IOTA 相关资料网址如下。

❑ 官网: <https://iota.org>

❑ 源码: <https://github.com/iotaledger>

❑ 白皮书: [https://iota.org/IOTA\\_Whitepaper.pdf](https://iota.org/IOTA_Whitepaper.pdf)

#### (1) DAG 结构

IOTA 的分布式账本 Tangle 基于 DAG 的数据结构。对于在 IOTA 网络上发生的每一笔交易, 都需要通过为它们进行微量的工作量证明 (PoW) 来批准前两笔交易。随着更多交易的发生, IOTA 网络能并行执行更多的验证, 因此网络将能够更快地扩展。

#### (2) Tangle 数据结构

在 Tangle 中, 每一个节点代表的是一个交易 (transaction)。如图 10-20 所示, 每个交易就是图中的一个顶点。当一个新的交易加入 Tangle 时, 它会选择 2 个先前的交易来批准, 为图形添加 2 条新的边。所以 DAG 是一种使用拓扑排序的有向图形数据结构, 由顶点和边



组成。它可以保证从一个顶点沿着若干边前进（有向），但永远不能回到原点（无环）。

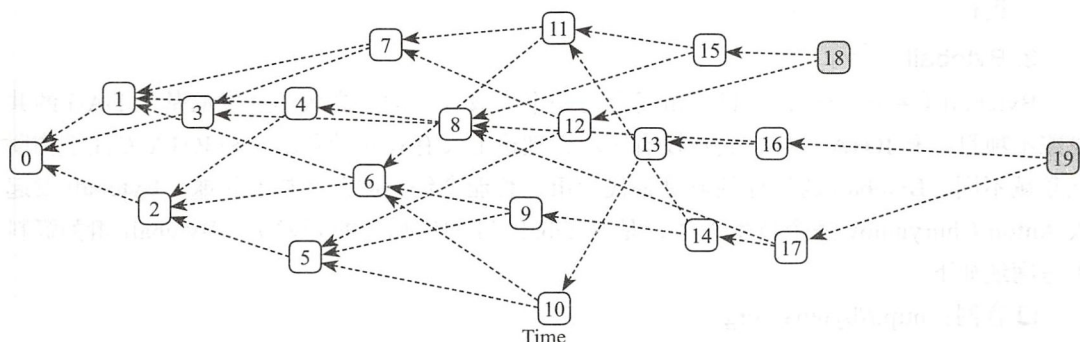


图 10-20 Tangle 图

### （3）共识机制

未批准的交易在 IOTA 里称为 tips。在图 10-20 中，交易号 18 和 19 就是 tips，因为还没有人批准它们。每个新加入的交易需要选择 2 个 tips 来批准，选择批准哪 2 个的技巧非常关键。IOTA 官方推荐用一种叫马尔可夫链蒙特卡洛（Markov Chain Monte Carlo, MCMC）的随机游走方法去选择这 2 个 tips。在马尔可夫链中，每一步都不依赖于前一个步骤，而是遵循预先决定的规则，这样能保证整个 Tangle 良好的发展。为了防止“双花”，IOTA 基金会采用了一种临时性的共识机制：协调员（Coordinator）机制。每隔 2 分钟，IOTA 基金会发布一项里程碑式交易，并且所有批准的交易立即被认为具有 100% 的确认信心（confirmation confidence）。具体选择 tips 的规则还没有发布，IOTA 基金会承诺当它成为法律实体之后会发布。我们可以假设发布的每个里程碑都会直接或间接地批准前一个里程碑。社区用户认为这样的共识机制有缺陷，不是真正的去中心化共识。IOTA 基金会承诺，日后当完整的 Tangle 分布式一致性算法开始奏效后，将关闭协调器，让 Tangle 完全依靠自身发展。

### （4）0 手续费

IOTA 采用的这种交易间相互验证确认的机制，使得交易互帮互助，不再需要为交易支付手续费。IOTA 有史以来第一次实现了 0 费用的转账，这意味着即使是无限小的微支付也可以通过 IOTA 来完成，使得智能终端之间根据设定的规则进行大量频繁的小额交易成为可能。

### （5）技术特性

- ❑ 数据结构：Tangle；
- ❑ 共识机制：简单 PoW；
- ❑ 哈希算法：Curl 三进制哈希算法；
- ❑ 交易验证策略：随机验证 2 个 tips，基于权重验证，MCMC 选择算法；
- ❑ 抗量子算法：基于 Winternitz 一次性签名技术的 Curl 算法（Curl 是量子安全的）；

□ 无网络依赖：除了 Internet，IOTA 可以在蓝牙、Zwave、ZigBee 或者 Lora 的环境下运行。

## 2. Byteball

Byteball（字节雪球），是目前市场上另一个大放异彩的、高效高并发的基于 DAG 的共享账本项目。和 IOTA 一样，它也没有区块，理论上没有吞吐量限制。和 IOTA 专注于物联网领域不同，Byteball 的目标领域是替代货币、智能合约、匿名和快速转账。Byteball 发起人 Anton Churymov 是个技术天才，是 Byteball AFAIK 唯一的开发员。Byteball 相关资料参考网址如下。

□ 官网：<http://byteball.org>

□ 源码：<https://github.com/byteball/byteball>

□ 白皮书：<https://byteball.org/Byteball.pdf>

### （1）DAG 结构

Byteball 是基于 DAG 结构的共享账本项目的又一优秀代表。和 IOTA 网络中每个节点代表一个交易（transaction）不同，Byteball 的 DAG 结构中每个节点代表一个存储单元（Unit）。一个单元含有多个数据包（信息），支持不同的信息格式，如交易信息、文本信息、简介、认证、资产、私人支付、固定面值资产、任意结构化数据、投票、私人消息等。Byteball 单元的 Hash 使用了 Merkle 树方式计算，由当前单元出发至创世单元的最长路径长度定义为单元级别。

### （2）共识机制：主链 + 见证人

传统的“区块 + 链式”的数据结构，需要有一个类中心化的操作，即“挖矿”来竞争记账资格，成功出块的矿工将获得奖励，并将当前所有交易验证打包到一个区块，然后发布到网络。而 Byteball 的 DAG 结构采用的是“单元 + DAG”结构，没有区块这一概念。所有单元由用户自己创建与发布，其验证与确认由引用其作为父单元的后辈单元来完成，可全网节点并发记录自己的单元数据，因而是一种更彻底、更高效的去中心化系统。与 IOTA 采用协调员 Coordinator 不同，Byteball 提出了创新的“主链”（MainChain）和“见证人”（Witness）的概念。从 DAG 的任意顶端（无子单元）开始，沿着最优父单元链接向创世单元进行，直至到达创世单元。这个从顶端到创世单元的链称为主链，如图 10-21 所示。见证人是非匿名、长期参与社区并拥有良好信誉的人，或是主动维护网络健康发展的组织。当前 Byteball 网络中设置了 12 个见证人，由见证人执行交易确认。每个交易单元会提供见证人列表。主链创造了一个全网共识确定的交易时间序列，优雅地避免了“双花”问题。

### （3）智能合约

IOTA 不支持智能合约，而 Byteball 拥有非图灵完备的声明式智能合约系统，采用 JSON 格式，支持简单逻辑计算脚本。这种智能合约语言能简单地直接描述合约期望的目标，表达能力强，易于理解，安全性高。与以太坊的智能合约相比，Byteball 的智能合约系统具有复杂度低、轻量化和高性能等优势，同时还降低了合约编写难度和出错概率，支持





### 3. TrustNote

TrustNote 针对现有区块链普遍存在的交易拥堵、交易费高、交易确认时间长等问题,以“极轻、极速、极趣”为目标,构建支持挖矿的 DAG 公有链,具有创新的双层共识机制和 TrustME 共识算法,支持海量并发交易且交易确认更快。TrustNote 聚焦于打造简单易用的去中心化数字通证底层区块链,利用增强表达能力的声明式智能合约,使用户可自由创建和发布数字通证,而无须编写复杂的智能合约代码。TrustNote 为数字通证、区块链游戏和社交网络提供安全的保障和丰富的应用接口,让新奇特的想法在区块链上流畅地运行。TrustNote 相关资料参考网址如下。

□ 官网: <https://trustnote.org>

□ 源码: <https://github.com/trustnote>

□ 白皮书: <https://github.com/trustnote/document>

#### (1) 双层共识机制

创新的双层共识机制,实现了高效率的同时满足了“去中心化”问题。第一层,基于 DAG 使得每一个区块链客户端能并行验证彼此之间的交易;第二层,通过挖矿确认超级节点,做出公证交易。

#### (2) 声明式智能合约

使用 JSON 形式的脚本声明式语言,支持条件判断和布尔运算。

#### (3) 节点类型

除了有 IOTA、Byteball 等其他 DAG 项目都设计采用的全节点和轻节点外,TrustNote 还支持物联网 IoT 友好的微节点,以及为工作量证明挖矿而生的超级节点。

#### (4) TrustMe 公证节点

解决 DAG 结构中低交易量时交易单元得不到及时确认的问题。

#### (5) 去中心化数字通证平台

模块化平台框架支持各种数字通证核心技术,以及各种类型的应用场景,如图 10-22 所示。

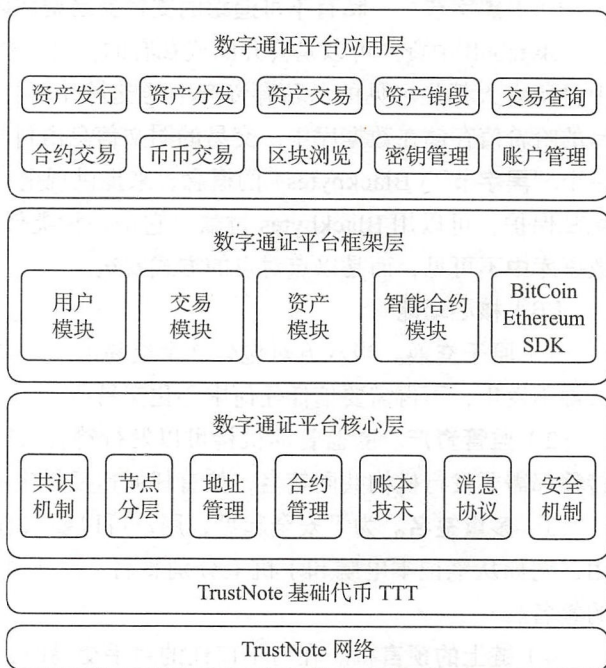


图 10-22 TrustNote 数字通证平台架构

### 10.3.9 Hashgraph: 区块链的新竞争者

Hashgraph 是 Swirlds 公司专利保护的一种快速、安全和公平的数据结构和共识算法。采用 Hashgraph 技术的项目 Hedera (常春藤) 在 2018 年下半年将要启动,使用“虚拟投票”



(virtual voting) 共识算法, 达成诚实度和透明度, 同时也节省 PoW 算法所费资源。它使用一种叫做“Gossip about Gossip”的协议, 记录“谁以什么顺序向谁 gossip 传播”, 创建一个网络交互的分布式账本, 使网络能够以极高的速度和效率进行扩展。目前采用 Hashgraph 技术的交易速度可达超过 250 000TPS。Hashgraph 相关资料参考网址如下。

□ 官网: <https://www.hederahashgraph.com>

□ 源码: 专利保护, 非开源, 但有一个免费的 SDK 包含几个演示应用程序及其源代码、平台文档以及如何编写应用程序的指导, 可以在 [www.swirls.com/download](http://www.swirls.com/download) 获得

□ 白皮书: <https://www.hederahashgraph.com/whitepaper>

### (1) 异步拜占庭容错

Hashgraph 也是 DAG 结构的一种。和同样采用 DAG 结构的 IOTA、Byteball 相比, Hashgraph 交易的最终确认方式实现了异步拜占庭容错, 故更加安全。

### (2) 共识算法

Hedera 的共识算法使用的“虚拟投票”(virtual voting), 网络中的每个成员都拥有 Hashgraph 的副本。如果两者都使用涉及发送投票的典型拜占庭协议, 允许甲方确定乙方会发送什么投票。在这种情况下, 乙方实际上并不需要投票, 所以每个成员都可以在没有投票的情况下达成一致。虚拟投票机制达成了一种新的诚实和透明度, 同时也节省了 PoW 算法所费的带宽, 如图 10-23 所示。

### (3) “Gossip about Gossip”协议

Gossip 是分布式系统中被广泛使用的协议, 其主要用于实现分布式节点或者进程之间的信息交换。一台计算机进行交易时, 随机告诉另外两台计算机。Hashgraph 创新了 Gossip about Gossip 协议, 调用任意随机节点并告诉该节点你知道的一切, 并在这个 Gossip 上附加一小部分信息, 其中包含我们刚刚交换(gossip)过的信息。这 2 台计算机告诉其他 4 台计算机, 并且数量会呈指数增长, 直到网络上的所有计算机都知道了这个交易。gossip 在网络上扮演着非常重要的角色。每个网络参与者都会对其学习的信息 gossip, 从而允许数据通过每个成员传播, 因为每个成员都会随机向其他成员反复传播信息。与现有的区块链协议不同, 这个协议使用非常少的带宽开销, 并防止网络膨胀。Gossip about Gossip 协议使网络能够以极高的速度和效率进行扩展。

### (4) 公平与速度

在区块链世界中, 矿工可以选择在一个区块中进行交易的订单, 可以通过将订单放置在未来的区块中来延迟订单, 甚至可以阻止它们进入系统。Hashgraph 网络通过交易促进公平, 所有成员都可以随时创建签名的交易, 并且每个成员都可以收到它的副本。社区根据这些交易的顺序达成协议。通过实现这种公平性, 一小部分攻击者不能恶意地影响选作共识的交易顺序。数学证明的公平性(通过共识时间戳)意味着没有人可以操纵交易的顺序。不像比特币网络每秒 6~7 笔、以太坊网络每秒 10 多笔的交易速度, Hashgraph 每秒可达到 250 000 以上次交易(预分片), 速度的快慢瓶颈仅受限于带宽。

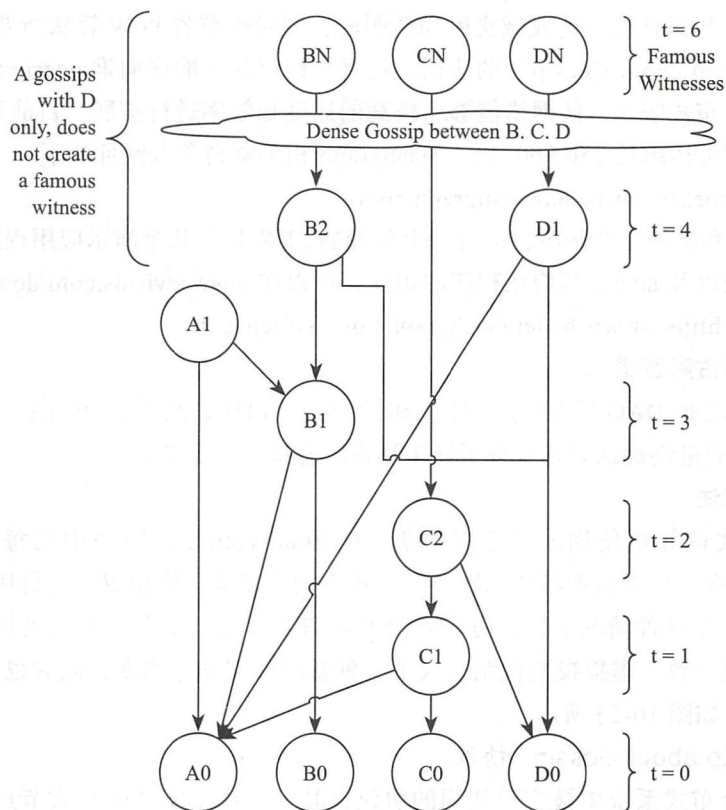


图 90-23 虚拟投票

### (5) 全球管理委员会

除了技术特质外，Hedera 的另一个独特之处在于其对企业监管的方法。Hedera 已经招募了来自包括大型科技公司和律师事务所在内的 39 家全球公司担任其“全球管理委员会”，负责重要战略决策，以增加合法性并帮助其获得主流支持。

### (6) 技术特性总结

- ❑ 数据结构：Hashgraph；
- ❑ 共识机制：virtual voting；
- ❑ 交易速度：250 000+ TPS；
- ❑ 安全机制：异步拜占庭容错；
- ❑ 共识协议：Gossip about Gossip。

## 10.3.10 区块链存储

大量数据如何存放在区块链上一直是一个问题。下面介绍几种分布式存储方案。





### 1. IPFS

IPFS (InterPlanetary File System), 星际文件系统, 使用对等超媒体分发协议, 将 IPFS 网络中的电脑通过协议连接成一个分布式文件系统, 资料网址如下。

❑ 官网: <https://ipfs.io/>

❑ 源码: <https://github.com/ipfs>

❑ 白皮书: <https://github.com/ipfs/papers/raw/master/ipfs-cap2pfs/ipfs-p2p-file-system.pdf>

Filecoin 是基于 IPFS 协议的去中心化的存储网络, 资料网址如下。

❑ 官网: <https://filecoin.io>

❑ 源码: <https://github.com/ipfs>

❑ 白皮书: <https://filecoin.io/filecoin.pdf>

IPFS 项目的雄心是想要颠覆 HTTP 模式。简单来讲, 用户在访问文件时, 不再基于域名地址的 URI 去寻找, 而是直接寻找这个文件内容的地址。这个地址就是 IPFS 系统生成的内容文件的哈希值, 通过这个唯一的哈希值就可以找到网络上存储了文件的电脑, 并快速下载。IPFS 的设计目标是把数据分割成碎片, 以去中心化的方式存储在参与节点里。它融合了一致性哈希 (DHT), 分布式版本管理 (Git) 和 P2P 协议 Bittorrent 的思想, 成为有望以基于内容寻址取代 HTTP 等基于域名寻址的 P2P Hypermedia 协议。IPFS 采用 Merkle DAG 来组织数据。2017 年基于 IPFS 的区块链存储项目 FileCoin 筹集 2.58 亿美元成为全球最大的 ICO 项目。FileCoin 支持智能合约, 采用 Proof-of-Storage 和 Proof-of-replication 共识机制, 支持代币交易。

### 2. Swarm

Swarm 是以太坊上的 P2P 文件共享协议, 提供分布式存储及分布式 CDN 功能。以太坊上的程序代码和数据存储在主链之外的 Swarm 节点。Swarm 节点与链上连接, 链上可交换数据。Swarm 可以共享存储和带宽, 内建点对点记账机制、激励机制、交易机制、参考资料网址如下。

❑ 官网: <https://twitter.com/ethersphere>

❑ 源码: <https://github.com/ethersphere>

❑ 白皮书: <https://github.com/ethersphere/swarm>

### 3. Sia

Sia 是由 Nebulous Labs 创建的分散存储网络。使用对等分散网络, 它创建了一个云存储市场, 其中一些用户充当主机, 在其磁盘提供可用空间, 而其他用户充当使用该存储空间的用户。Sia 支持智能合约来制定存储的规则和需求, 用户之间可买卖存储空间。Sia 的最终目标是成为互联网的骨干存储层, 创建一个分散的数据中心网络, 参考资料见以下网址。

官网: <https://sia.tech>



源码: <https://github.com/NebulousLabs/Sia>

白皮书: <https://sia.tech/sia.pdf>

#### 4. MaidSafe

MaidSafe 定位“众包互联网”，提供 Marketplace 和代币交易。采用资源拥有证明 POR (Proof Of Resource) 的方式，MaidSafe 用户存储数据可以通过彩票形式随机获得 Safecoin 代币，获得 Safecoin 的多少与提供的资源和开机时间相关，参考资料见以下网址。

❑ 官网: <https://maidsafe.net>

❑ 源码: <https://github.com/maidsafe>

❑ 白皮书: <http://docs.maidSAFE.net/Whitepapers/pdf/Safecoin.pdf>

#### 5. Storj

Storj 文件和数据分片加密，存储到多个节点。Storj 不支持智能合约，其代币 SCJx 支付给贡献存储的存储节点，它可以像“燃料”一样允许用户在名为“DriveShare”的 App 中使用，通过 SJCX 来租用或者购买存储空间，就像 MetaDisk 一样，参考资料见以下网址。

❑ 官网: <https://storj.io>

❑ 源码: <https://github.com/storj>

❑ 白皮书: <https://storj.io/storj.pdf>

#### 6. BurstCoin

BurstCoin 基于 NXT 区块链，支持智能合约，采用 Proof-of-Capacity(容量证明)共识算法，由每个矿工计算生成一个大的存在矿工节点的数据集，每个新的区块中，每个矿工将读一个数据集的小部分 (0.024%) 并返回一个时间 (Deadline)，即为上个区块生成到现在允许生成新区块的时间，用时最短者获得产生区块权 BurstCoin 相关参考资料网址如下。

❑ 官网: <https://www.burst-coin.org>

❑ 源码: <https://github.com/burst-team>

❑ 白皮书: <http://www.burst-coin.org/wp-content/uploads/2017/07/The-Burst-Dymaxion-1.00.pdf>

#### 7. 移动区块链

OFBANK (以下简称 OF)，中文翻译为 OF 社群链。OFBANK 团队正在开发设计一条基于全新共识且运行在移动终端上的区块链，其全节点同步在真实用户的手机上。

OF 移动区块链的设计思路认为，移动互联网与 PC 互联网的本质不同在于，PC 互联网调动的是机器的计算和数据存储与传输，移动互联网调动的是人以及每一个人背后的资源，包括机器、工业和整个人脉的能量。比特币兴起是因为社群的推动，矿机只是一个过渡阶段的产物，很快会被超级计算机和手机取代。未来真正有生命力的应该是移动区块链，因





为移动区块链更能体现区块链对等、去中心精神的精髓。

基于手机的移动生活已经成为人们生活不可分割的一部分,移动用户通过手机形成共识、执行智能合约,是移动区块链的基本需求。

### (1) PoA 共识机制

当前比特币等区块链采用 PoW 共识机制,实际上 PoW 的算力已经出现大量的冗余,相对来说这是一种社会资源浪费。OFBANK 团队正在设计一种命名为 PoA 的手机共识机制,即用户贡献证明。通过移动终端不同的用户贡献(如浏览时间、电商交易、社交贡献、游戏工作量等),结合部分手机算力与用户股权,形成一种真实有效的用户挖矿行为,以获得数字积分的贡献激励,并且该系统不断地自我进化,迭代挖矿激励算法,最终实现一种动态平衡。

OF 以社群替代矿池的方式组织用户,这种用户行为贡献的模式,可以称之为一种“挖矿”。OF 将用户信任、奖励机制和区块链共识有机地结合起来,把每一个社群和应用看作一个“矿池”,用户类比为“矿工”,用户对社群的贡献,就是对系统的贡献(挖矿),从而得到代币奖励。这样使整个生态能够自生长、自进化,以社会化的方式生生不息地运转下去。

### (2) 安全结构

手机本身的用户分布,是典型的 P2P 网络,通过手机用户实现的共识机制,在面对安全攻击时,一样符合帕松分布(Poisson Distribution),也就是说,在单位时间内,安全攻击事件发生的概率分布呈帕松分布。OFBANK 团队为移动区块链设计了新的安全结构,其原理与 PoW 异曲同工,简单来说,攻击 1000 万台矿机的难度和攻击 1000 万部手机(移动用户)的难度是一样的,因此其系统安全性和公正性也是可靠的。

### (3) 轻量级的虚拟机

运行该移动区块链,需从底层实现一套轻量级的全新的虚拟机系统,用来在手机上运行智能合约。其技术突破点在于:降低内存开销、增加执行效率;编译后的字节流可以跨平台运行;该虚拟机亦完全遵守沙盒机制,防止网络中的恶意攻击;让用户可以在手机端发布智能合约。

### (4) 精细化智能合约

在 OFBANK 手机区块链上,将有无数合约工程师来设计颗粒度无限放大的智能合约,从而解决复杂社会下复杂网络的各种问题。其中见证算法将发挥重要价值,基于人与人协作的共识机制,将智能合约真正落地于大众生活,无形但无处不在。

### (5) DApp 开发

基于移动区块链封装完善的 SDK,能更快实现 DApp 的开发和部署。

### (6) 链群及数字积分流通

OF 移动区块链设计了侧链链群模型,基于双向锚定原理,OFBANK 的双链结构及侧链之间,数字积分能够互相流通。在 OF 的侧链系统中,正在运行的任何一条链都可以认为是一条主链,其他链为侧链,这种对等关系的互为主侧链的链群结构,是一种类“榕树生



长”模型，对生态孵化意义重大。

### (7) 发展现状

目前基于模拟算法的 OF 手机挖矿 DApp，4 个月已经获得 150 万用户，OFBANK 已成为中国 C 端用户最多的公有链。

## 10.3.11 安全和隐私保护

安全和隐私一直是区块链中核心技术的重要组成部分。在这方面有所创新的加密货币代表有以下 3 种。

1) 门罗币 (Monero, 缩写 XMR): 采用环签名方式来保护交易方隐私。

□ 官网: <https://getmonero.org/>

□ 源码: <https://github.com/monero-project>

□ 白皮书: <https://getmonero.org/resources/research-lab>

2) 达世币 (Dash): 采用 X11 哈希算法来保证密码强度。

□ 官网: <https://www.dash.org>

□ 源码: <https://github.com/dashpay>

□ 白皮书: <https://github.com/dashpay/dash/wiki/Whitepaper>

3) 零币 (ZCash): 采用非交互式零知识证明 (zk-SNARKs) 来提供匿名和交易细节隐藏。

□ 官网: <https://z.cash>

□ 源码: <https://github.com/zcash>

□ 白皮书: <https://z.cash/technology>

以上几个项目我们不详细介绍，大家有兴趣可以通过上述链接自行研究学习。其他安全隐私手段还有代码混淆 (Code Obfuscation) 以及用 Elliptic Curve Diffie-Hellman-Merkle (ECDHM) 地址来隐藏真实地址。另外提供完全隐私保护的完全同态加密、抗量子攻击密码算法也是科研界和区块链社区积极研究的安全方向。

## 10.4 一句话解释主要加密货币

网上流行用不多于 4 个英文单词来解释主要加密货币的活动，笔者也用一句话来解释主要加密货币，以促使我们对各种主流加密货币有个简明扼要且直观的认识。

Bitcoin | BTC: 数字黄金

Ethereum | ETH: 智能合约和电子现金

Bitcoin Cash | BCH: 比特币克隆

Ripple | XRP: 企业转账清算网络

Litecoin | LTC: 更快版本的比特币

Dash | DASH: 更注重隐私保护的比特币克隆





NEO | NEO: 中国版以太坊

NEM | XEM: 自带电池的数字资产

Monero | XMR: 匿名数字现金

Ethereum Classic | ETC: 以太坊克隆版

IOTA | MIOTA: 物联网转账支付

Qtum | QTUM: 比特币上的智能合约

OmiseGO | OMG: 银行、汇款、交易所

Zcash | ZEC: 匿名数字现金

BitConnect | BCC: 类似于麦道夫投资基金

Lisk | LSK: 用 JavaScript 编写的分布式应用

Cardano | ADA: 分层数字现金和智能合约

Tether | USDT: 锚定 1 美元

Stellar Lumens | XLM: 数字现金 IOU 欠条

EOS | EOS: WebAssembly 上的分布式应用

Hshare | HS: 区块链交换机

Waves | WAVES: 分布式交易所和众筹

Stratis | STRAT: C# 版的分布式应用

Komodo | KMD: 分布式 ICO

Ark | ARK: 区块链交换器

Electroneum | ETN: 门罗币克隆

Bytecoin | BCN: 匿名数字货币

Steem | STEEM: 用代币投票的 Reddit 论坛

Ardor | ARDR: 可以生成区块链的母链

Binance Coin | BNB: 抵偿币安交易费

Augur | REP: 分布式预测市场

Populous | PPT: 区块链票据金融系统

Decred | DCR: 拥有自主管理机制的比特币

TenX | PAY: 加密货币信用卡

MaidSafeCoin | MAID: 出租硬盘空间

BitcoinDark | BTCD: Zcoin 克隆

BitShares | BTS: 分布式交易所

Golem | GNT: 出租算力

PIVX | PIVX: 不会通胀的 Dash 币克隆

Gas | GAS: Neo 的转账支付费

TRON | TRX: App 内支付



Vertcoin | VTC: 比特币克隆

MonaCoin | MONA: 日本狗狗币

Factom | FCT: 分布式数据记录

Basic Attention | BAT: 分布式广告网络

SALT | SALT: 加密货币抵押网络

Kyber Network | KNC: 分布式交易所

Dogecoin | DOGE: 搞笑版比特币克隆

DigixDAO | DGD: 由公司管理的黄金数字货币化

Veritaseum | VERI: 雾件

Walton | WTC: 物联网区块链

SingularDTV | SNGLS: 分布式 Netflix

Bytom | BTM: 作为通证的实物资产

Byteball Bytes | GBYTE: 分布式数据库和货币

GameCredits | GAME: 电子游戏货币

Metaverse ETP | ETP: 中国版以太坊和身份证

GXShares | GXS: 分布式中国版 Equifax

Syscoin | SYS: 分布式市场

## 10.5 小结

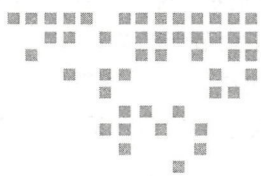
本章我们分析了当前区块链构架存在的问题和挑战；研究了区块链平台的典型需求，思考了其主要的发展方向；简单介绍了其他一些富有特色的区块链平台以及它们的特点和应用。通过本章的学习，读者应该较为全面地了解了当前区块链的技术挑战和应用创新难题，学会了判断哪些是真正有潜力的行业和技术研究发展方法。我们特意给出了许多项目白皮书以及源代码的资源链接，供读者根据需要，结合自身实际情况进一步深入学习研究。由于区块链行业还处在发展早期，技术发展日新月异，各类项目迭代变化很快，笔者时间和见闻有限，故难免有所疏漏和偏颇，请读者指正。

### 习题

- [1] 什么是区块链三难选择？
- [2] 请简要列举当前区块链平台的问题和痛点。
- [3] 区块链平台有哪些技术需求？
- [4] 什么是侧链技术，有哪些代表性项目？
- [5] 什么是分片技术，有哪些代表性项目？
- [6] 为什么说 DAG 是区块链 3.0 技术的强有力竞争者？
- [7] 区块链上交易隐私有哪些保护措施和方法？







## 区块链常见问题剖析

作者：于斌

### 11.1 区块链的技术局限

#### 11.1.1 区块链不可能三角

在第 10 章时，我们提到了区块链不可能三角（Blockchain Trilemma）理论，也就是说，区块链系统不可能同时达到完全去中心、安全性、扩展性 3 个目标。那具体怎么理解呢？

所谓“完全去中心”，指的是网络中每一个节点都可以访问  $O(c)$  量级的资源。 $c$  是系统中每个节点能拥有的计算资源，包括算力、带宽和存储， $O$  是表示复杂度的符号， $O(c)$  的意思是指复杂度相当于  $c$  乘上某个常量。也就是说，节点在网络中是完全对等的，不存在一个中心化的节点垄断资源。安全性指的是能够保护  $O(n)$  多个资源。 $n$  是一个抽象的生态系统里的大小。而扩展性指的是能够处理  $O(n)$  个交易，这里要求  $O(n) > O(c)$ 。简单来理解，就是如果要能够处理  $O(n)$  个交易，同时保护  $O(n)$  个资源，那么只能集中在少数节点上，否则会带来资源的竞争和受攻击面的扩大。

目前的区块链平台，还没有一个能同时达到在完全去中心、安全性和扩展性目标。比特币的设计是为了提供一个安全的去中心化电子现金系统，因此它牺牲了交易性能。以太坊和比特币类似，也采用完全去中心化的设计，它比比特币要快，但安全性相对要差一些。而 EOS、BTS 等交易速度更快，但去中心化程度却大大降低，成为部分去中心的系统。

#### 11.1.2 数据冗余

区块链的内在冗余机制，使其具有健壮性和高容错的优点。每个全节点都保存一份完整的区块数据，即使某些节点出错，也不会导致整个网络垮掉。同时宕机的节点，重启之



后也总是能正确同步新的交易。但另一方面，数据确实又比传统中心化系统冗余了  $n$  倍，其中  $n$  是网络中全节点的数量。

所以说区块链其实是一项应用起来有一定成本的技术。如果应用不能通过区块链创造比中心化体系更多的价值的话，其实是不适合用区块链技术的。

### 11.1.3 区块链安全性

另一个常见问题是，在区块链中，数据安全吗？首先我们声明一下，在这个世界上就没有绝对的安全，只有相对的安全，矛和盾的关系永远存在。

可以说，区块链是现在互联网安全的升级版，有着比传统的互联网更高的安全性。区块链是比特币中的底层技术，在传统的无法建立信任关系的互联网上，区块链技术依靠密码学、分布式共识算法和巧妙设计的激励机制，无须借助任何第三方中心机构的介入，用数学的方法使参与者达成共识，保证交易记录的存在性、合约的有效性以及交易的不可抵赖性，大大提高了系统的安全程度。

但是基于区块链的数字货币的火热行情让黑客们垂涎不已，被盗金额不断刷新纪录。盗窃事件的发生也引发了人们对数字货币安全的担忧？

随着人们对区块链技术的研究与应用，区块链系统除了其所属信息系统会面临病毒、木马等恶意程序威胁及大规模 DDoS 攻击外，还将由于其特性而面临独有的安全挑战。

主要体现在以下几个方面。

#### （1）算法实现安全

由于区块链大量应用了各种密码学技术，属于算法高度密集工程，在实现上比较容易出现问题。历史上有过此类先例，比如美国的 NSA 对 RSA 算法实现埋入缺陷，使其能够轻松破解他人的加密信息。一旦爆发这种级别的漏洞，可以说构成区块链整个大厦的地基将不再安全，后果极其可怕。之前就发生过由于比特币随机数产生器出现问题所导致的比特币被盗事件。理论上，如果随机数产生不够随机，就能通过碰撞推导出私钥。

#### （2）共识机制安全

当前的区块链技术中已经出现了多种共识算法机制，最常见的有 PoW、PoS、DPoS。但这些共识机制是否能实现并保障真正的安全，需要更严格的证明和时间的考验。

#### （3）区块链使用安全

区块链技术一大特点就是链上数据不可篡改、不可伪造，但前提是私钥是安全的。私钥是用户生成并保管的，理论上没有第三方参与。私钥一旦丢失，便失去与之相对应的数字资产。一旦被黑客拿到，就能转移相应的数字资产。目前绝大多数的安全问题都出现在私钥被盗上面。

#### （4）系统设计安全

某些平台由于在业务设计上存在单点故障，其系统容易遭受 DoS 攻击。目前区块链是去中心化的，而大多数交易所是中心化的。中心化的交易所，除了要防止外部黑客盗窃外，





还得管理好内部员工，防止作恶的内部员工盗窃。另外系统的业务设计缺陷也可能被黑客利用。

那么如何保证区块链的安全？

为了保证区块链系统安全，建议参照 NIST 的网络安全框架，从战略层面、一个企业或者组织的网络安全风险管理的整个生命周期的角度出发，构建识别、保护、检测、响应和恢复 5 个核心组成部分，来感知、阻断区块链风险和威胁。重点关注算法、共识机制、使用及设计上的安全。

1) 针对算法实现安全性：一方面选择采用经过长期验证过的密码技术；另一方面对核心算法代码进行严格、完整测试的同时进行源码混淆，增加黑客逆向攻击的难度和成本。

2) 针对共识算法安全性：使用更有效的共识算法和策略。

3) 针对使用安全性：对私钥的生成、存储进行保护，对用户进行信息安全教育，对敏感数据加密存储。

4) 针对设计安全性：一方面要保证设计的功能尽量完善，如采用私钥白盒签名技术，防止病毒、木马在系统运行过程中提取私钥，设计私钥泄露追踪功能，尽可能减少私钥泄露后的损失。另一方面，应对某些关键业务设计成去中心化系统，防止单点故障攻击。

### 11.1.4 挖矿和其他共识协议的弱点

每个共识协议都有弱点，下面我们分几个主要的共识机制来谈这个问题。

1) PoW 工作量证明，就是大家熟悉的挖矿，通过哈希运算，找到一个满足规则的随机数，即获得本次记账权，发出本轮需要记录的数据，全网其他节点验证后一起存储。

□ 优点：完全去中心化，节点自由进出；

□ 缺点：目前比特币已经吸引全球大部分的算力，其他再用 PoW 共识机制的区块链应用很难获得相同的算力来保障自身的安全；挖矿造成大量的资源浪费；共识达成的周期较长，不适合商业应用。

我们以比特币为例。在比特币的挖矿系统中，能源消耗是个大问题。比特币网络和一般的 P2P 网络是不同的：一般 P2P 网络节点间相互补充和合作，比特币网络的 P2P 节点之间相互竞争和重复。无论转账量多大，每个比特币网络节点都必须独立完成所有转账运算量，并通过进行无意义的哈希计算来竞争获取账本写入权和挖矿奖励。也就是说，这是一种代价和浪费巨大的 P2P 网络。相比传统金融而言，其运作成本将高出几个数量级。目前比特币的转账量不过十分钟数百次，但其一年耗费的电量已经可以和一个像爱尔兰这样的小国家一年消耗的电力相提并论了。

由于比特币的挖矿奖励币数每 4 年减半，未来当挖矿的比特币接近枯竭，矿工要求的手续费必将高涨。否则矿工就会失去动力来保证比特币网络的安全；而高昂的手续费又可能会让想使用比特币的用户望而却步，因此比特币网络未来的可持续运行可能是个问题。

2) PoS 是不同于 PoW 的一种共识机制。根据每个节点所占代币的比例和时间，等比



例的降低挖矿难度，从而加快找随机数的速度。

□ 优点：在一定程度上缩短了共识达成的时间。

□ 缺点：存在作恶节点低成本分叉的可能性，因此安全性比 PoW 低。另外由于获得记账权的几率和持币量相关，而币又是一次性发出、一次性分配，因此公平性比 PoW 差。

3) DPoS 股份授权证明机制。类似于董事会投票，持币者投出一定数量的节点，代理进行验证和记账。

□ 优点：大幅缩小参与验证和记账节点的数量，可以达到秒级的共识验证。

□ 缺点：整个共识机制的中心化程度比单纯的 PoS 要高。

4) Paxos 或 BFT 类强一致共识算法。基于传统的分布式一致性技术，加上数据验证机制，是目前行业链大范围在使用的共识机制。

□ 优点：不需要代币也可以工作，在成熟的分布式一致性算法（Paxos、Raft）基础上，实现秒级共识验证；

□ 缺点：去中心化程度不如比特币；更适合多方参与的多中心商业模式。另外扩展性也有限，当共识节点数增加时，共识性能下降很快。

### 11.1.5 交易速度

区块链技术由于使用了分布式冗余记账技术，再加上各节点进行状态共识，因此会牺牲系统的速度。使得交易速度成为诟病区块链的一个重要因素。区块链交易处理速度是投入商业应用的重要考量之一。

衡量区块链每秒交易量的一个重要指标就是每秒可完成的交易量（TPS）。其计算公式为：

$$\text{TPS} = \text{区块平均交易数} / \text{出块间隔}$$

以最常用的 PoW（PoS）为例。交易量是有上限的，这个上限取决于互联网目前的网速和计算机的计算和响应速度。这是因为无论是 PoW 还是 PoS，都需要根据这些给信息传输和处理以时间。单纯提高区块大小或者减少区块生成之间的时间不能有效解决问题，因为区块需要时间传输验证。如果区块太大，就会造成网络节点的不一致性加重（分叉变多），从而严重影响安全性和可靠性。所以，PoW 或者 PoS 的区块链，交易很难超过 100tx/s 这个量级。

解决的方法，除了提高算法效率和网络传输速度，基本上都是以牺牲去中心化程度以及安全性和可靠性为代价来增加输出。这并不是一件坏事，因为实际上当前的很多共识算法的安全性和可靠性假设超过了实际需求。

要想提高每秒交易量，从理论上来说有以下方法。

1) 提高区块容量。BCH 就是实例（注意，BCH 的区块大小为 8MB），目前 BCH 如果写满每一个区块，一年就会产生约 400GB 的区块链数据。与比特币相比数据量会很大。

2) 降低出块间隔。缺点就是一定程度上会牺牲安全性。



3) 降低交易字节大小。在比特币设计中,一笔交易大小约为 250 ~ 400 字节,如果能够降低每一笔交易的字节大小,就可以使一个区块内记录更多笔交易。一笔交易的记录包含了交易记录和签名信息,签名信息要比交易信息大很多,而且签名信息只有矿工在将交易打包入块,验证合法性时才有用,对于非矿工来说其实是没有用的。所以比特币的隔离见证实际上是把签名信息移到交易外部,不计入区块大小判断来间接提升交易数量。但以上这些方法都只能线性提高每秒交易量。

共识机制的改变能从根本上把 TPS 提高到过万这个级别, BTS 可以说是一个很好的例子。

还是以比特币为例。现在提高比特币每秒交易量的方案有:①硬分叉,即增大区块大小;②软分叉,即优化网络但不改变区块大小(还是 1MB),如隔离见证、闪电网络、侧链等。

## 11.2 区块链的安全问题

### 11.2.1 51% 攻击问题

我们常常听到的 51% 攻击是什么?

其实问题的核心就是要有超过一半以上的算力就可以“双花”链上的资产。下面我们举例说明。如果我们想象十个人中的一个人试图欺骗和修改区块链的内容,他将不得不调整几个区块并计算新的符合难度目标的哈希值。我们知道要找到符合条件的哈希值非常困难,因此,网络上一个不诚实的人无法击败九个诚实的人。

如果一个不诚实的人试图在一个区块上做出欺骗行为,那会在网络中创造另一个链条,但是欺骗的链条永远无法赶上诚实的链条——因为一个人的努力和速度无法击败九个人累积的努力和速度。因此,这个机制可以保证网络中最长的链是最诚实的链。

但如果不是一个人不诚实,而是有六个人不诚实呢?在这种情况下,篡改区块链上的记录就可以实现。这被称为“51%攻击”。如果网络中的大多数人都变得不诚实,并欺骗网络的其余部分,协议目的就失效了。但是我们必须知道区块链系统的弱点:它建立在大多数人群总是诚实的假设之上。

### 11.2.2 女巫攻击

大规模的 P2P 系统面临着有问题的和敌对的节点的威胁。为了应付这种威胁,很多系统采用了冗余。然而,如果一个有恶意的实体模仿了多个身份,他就可以控制系统的很大一部分,破坏系统的冗余策略。我们把这种模仿多个身份的攻击定义为“女巫攻击”(Sybil Attack)<sup>①</sup>。在女巫攻击中,攻击者通过创建大量的假名标识来破坏对等网络的信誉系统,使用它们获得不成比例的大的影响。

---

① “女巫攻击”的名字由 Brian Zill 建议,来源是 20 世纪 70 年代一部叫做《Sybil》的美国系列片。片中的女主角人格混乱,扮演着 16 个角色。

对等网络上的实体是能够访问本地资源的一个软件。实体通过呈现身份在对等网络上通告自身。多于一个标识可以对应于单个实体。换句话说,身份到实体的映射是多对一的。对等网络中的实体为了冗余、资源共享、可靠性和完整性而使用多个标识。

默认情况下,通常假定每个不同的标识对应于不同的本地实体。实际上,许多身份可以对应于相同的本地实体。这样攻击者就可以向对等网络呈现多个身份,充当多个不同的节点,并可能能够获得对网络的不成比例的控制水平,例如影响投票结果。

女巫攻击包括以下类型。

- 1) 直接通信:女巫节点直接与合法节点进行通信。
- 2) 间接通信:没有一个合法的节点能够直接与女巫节点进行通信。相反,一个或多个恶意的节点宣称它们能够到达女巫节点。就是这个恶意节点自己接收或者拦截相关消息。
- 3) 伪造身份:攻击者可以产生任意的女巫身份,以进行攻击。
- 4) 盗用身份:攻击者进行节点的身份盗用,在原有节点摧毁或者失效前很难检测。
- 5) 同时攻击:攻击者将其所有的女巫身份一次性地同时参与到一次网络通信中。攻击者可以循环使用它的多个女巫身份,让人看起来是多个节点。这就是同时性。
- 6) 非同时攻击:攻击者只在一个特定的时间周期里使用一部分女巫身份,而在另外一个时间段里使这些身份消失而以另外的女巫身份出现。这看起来就像网络中正常的节点撤销和加入。

传统防止女巫攻击的方法是采用一个信任的代理来认证实体。

验证技术可用于防止女巫攻击和消除伪装敌方实体。本地实体可以基于中心化权威机构来验证远程身份,其确保身份和实体之间的一一对应,甚至可以提供反向查找。身份可以直接或间接地验证。在直接验证中,本地实体查询中央授权机构以验证远程标识。在间接验证中,本地实体依赖于已经接受的身份,继而保证所讨论的远程身份的有效性。

这种办法不适用于去中心化程度高的公有链。在像以 PoW 为共识机制的区块链,例如比特币或以太坊,采用算力为获得记账权的依据,因此多节点的女巫攻击失去意义。而像以 PoS 为共识机制的区块链,例如点点币,采用拥有的币龄为获得记账权的依据,同样,多节点的女巫攻击也失去作用。

### 11.2.3 交易所

目前的数字交易所绝大多数是中心化的,其安全性隐患非常大,基本上可以包括人为的安全性和技术系统的安全性漏洞。这些交易所因为储存了大量的加密货币而成为黑客的觊觎目标。以下是几次著名的交易所安全事件,说明数字交易所存在严重安全隐患。

- 2018 年 3 月,全球第二大交易所币安遭遇黑客入侵,导致大量用户账户里的比特币被卖掉,买入一虚拟币;黑客同时在世界各大交易所挂比特币空单套利。虽然黑客没有直接从币安提币,但却造成币安被侵入账户的账面损失,并间接造成了大量比特币持有者的账面损失。



❑ 2018年1月,日本最大的加密货币交易所Coincheck遭黑客袭击,价值5.3亿美元的数字货币不翼而飞。

❑ 2017年12月,韩国比特币交易所Youbit因遭遇“黑客攻击”,丢失了17%的数字货币,宣布破产。

基本所有交易所系统都面临技术上的安全隐患。有以下几类安全性问题:

- ❑ 系统安全;
- ❑ 硬件设备安全;
- ❑ 交易通道安全;
- ❑ 数字钱包安全;
- ❑ 终端安全;
- ❑ 用户操作安全。

同时也有许多项目开始研发运行在区块链的去中心化交易所。比起中心化的技术,运行在区块链上的去中心化交易所是利用区块链的三个特性——分布式、透明、不可篡改的特性来加强交易所的安全性与透明度。

### (1) 中心化交易所

中心化交易所(Centralized Exchange)其实就是用户主要使用的交易所,Bitfinex、Poloniex、Coincheck、币安等都是中心化交易所。用户使用这些交易所的方式,通常就是到网站上注册,根据不同国家的法规经过一连串认证程序后,把加密货币转入他们指定的钱包地址后,就可以开始在上面交易加密货币。

其中的交易不见得会发生在区块链上真正的货币交换,取而代之的可能仅是修改交易所数据库内的资产数字,用户看到的只是账面上数字的变化,交易所只要在用户提款时准备了充足的加密货币汇出即可。

当使用者把加密货币转到他们提供的钱包地址后,交易所就拥有了操作这些加密货币的权利,使用者必须要信任这个网站会保证货币安全,才能把加密货币转给交易所操作。

正因为中心化交易所拥有了存放大量加密货币的私钥,所以非常容易吸引黑客的攻击。而黑客的目标绝大部分就是这些存放大量加密货币的私钥,偷走了这些私钥就代表盗走这些加密货币。

如图11-1所示为中心化交易所技术架构。

### (2) 去中心化交易所

去中心化交易所(Decentralized Exchange)跟一般中心化交易所最不一样的地方,就是交易行为发生在区块链上。与中心化交易所的交易实质是在交易所本身的数据库中增减用户资产字段不同的是,去中心化交易所是在区块链上直接交换,加密货币会直接发回使用者的钱包,或是保存在区块链上的智能合约。

这样直接在区块链上交换的好处在于,交易所并不持有用户大量的加密货币,所有的加密货币会储存在区块链上使用者的钱包或智能合约控管。本来需要信任中心化的交易所,

现在仅需要信任区块链以及智能合约即可。而用于交易所的智能合约大多会公开源码让所有人可以确认这份合约的细节。

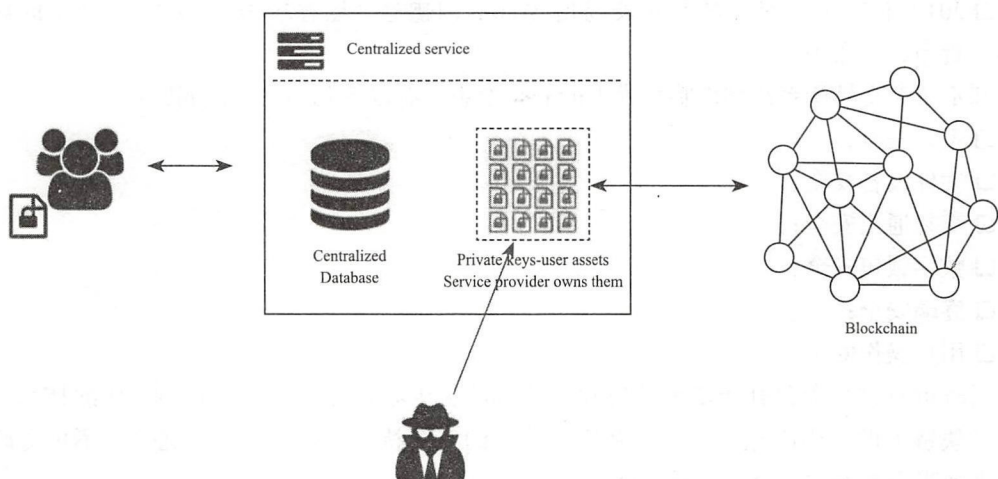


图 11-1 中心化交易所技术架构

而中心化交易所通常不会公开交易细节的源码，实际上如何运作也只有其内部人员知道。通过公开的智能合约维持交易过程的透明与安全性，有利润的拆分也会公开在区块链上。相信用户会因为信任问题逐渐转向去中心化的交易所。

目前世界上已经有了几种去中心化的交易所在运行，比如 EtherDelta 和 Kyber Network 都是比较早期开始发展的交易所。由于开发运营的时间较短，它们尚不够成熟，有待逐渐完善。与中心化交易所相比，去中心化交易所的一个弱势是没有撮合引擎，成交效率低。

#### 11.2.4 以太坊智能合约安全漏洞

传统的计算机程序一般运行在某个节点或集群上，为某个机构或个人拥有和控制。这样的计算机程序可以随时施加人工干预，可以随时控制。然而，区块链上的智能合约则是一种全新的计算范式。智能合约一经部署就难以修改，其执行也是自动执行，不受人为了干预。因此，如果智能合约有安全漏洞，就很难防范黑客的攻击。当下，专业技术人员都在努力寻找合适的方式，提高智能合约的安全性和可靠性，帮助保障大家的资金安全。根据一份针对以太坊智能合约的最新调查显示，在安全问题研究人员看来，智能合约属于新兴事物，所以缺陷和漏洞还是非常多的。

所谓“智能合约”，其基本内涵就是指可以将一些带有合约性质的条款嵌于我们日常使用的硬件和软件当中，目的是让那些违约者在违反合同时付出一定代价。Szabo 将实体售货机称为“智能合约的原型”，因为实体售货机就是根据屏幕上所显示的价格，收取用户投进来的硬币，掉出用户所选中的商品并且正确找零。

以以太坊为例。以太坊是一个开源的有智能合约功能的公共区块链平台。区块链上的



所有用户都可以看到基于区块链的智能合约。但是,这会导致包括安全漏洞在内的所有漏洞都可见。如果智能合约开发者疏忽或者测试不充分,而造成智能合约的代码有漏洞的话,就很容易被黑客利用并攻击。并且越是功能强大的智能合约,逻辑就越复杂,也越容易出现逻辑上的漏洞。同时,智能合约语言 Solidity 自身与合约设计都可能存在漏洞。

以太坊开源软件主要是由社区的极客共同编写的,目前已知存在 Solidity 漏洞、短地址漏洞、交易顺序依赖、时间戳依赖、可重入攻击等漏洞。在调用合约时漏洞可能被利用,而智能合约部署后难以更新的特性也让漏洞的影响更加广泛持久。

另外,以太坊虚拟机(EVM)对于智能合约能够做的事情存在很多硬性限制。这些都牵扯到平台级的安全,甚至可能会威胁到用户特定合约的安全。

伦敦大学学院(University College London)的计算机科学家 Ilya Sergey 表示,目前专业技术人士还没有完全搞清楚智能合约中存在的安全漏洞和潜在风险。他曾经参与过一项针对智能合约的调查研究。Sergey 及其同事借助一款创新工具,对将近 100 万份的以太坊智能合约样本进行了分析。结果发现,其中约有 3.4 万份都是存在安全隐患的,包括导致 Parity 事件的那一份。Sergey 介绍说,自己和团队其他成员的分析工作,就好比是与自动售货机互动。研究人员随机按下某个按钮,接着记录机器在运行过程中出现意外事故时的相关状况。用他的话说:“我认为,目前我们没有发现的安全漏洞还有很多,需要继续分析并且进行分类。”

在实践中如何提高智能合约的安全性?

给出以下建议,在实际编程中尽量遵守,你的合约将更具安全因素。

- 1) 更完善的编写测试。
- 2) 建议提供容错和自动错误赏金。
- 3) 为最糟糕的情况做准备。智能合约中的漏洞,应该尽可能让它安全地恢复。
- 4) 添加额外的安全机制。合同的管理者可应急性地冻结合约。
- 5) 限制合约资金存放金额,提高攻击者成功的门槛。
- 6) 不要从零开始编写你所有的代码,尽可能参考成功者的合约。
- 7) 注意开发平台的限制。

### 11.2.5 区块链安全性的测试指标

区块链的安全性主要考察身份验证、访问控制、加密体系和隐私、密码算法、匿名性、抗攻击能力 6 个方面。以下是具体的测试指标分析。

- 1) 身份验证。身份验证主要测试 5 个指标:身份验证的方式、身份验证的场景、防止身份冒用、私钥管理、节点的进出需要身份验证。
- 2) 访问控制。访问控制主要测试 7 个指标:不同级别账户权限不同、不同分类账户权限不同、超级账户的权限、账户间的授权、账户授权时限控制、账户权限变更、权限变更形式。
- 3) 加密体系和隐私。加密体系和隐私主要测试 4 个指标:交易信息加密、签名的应用

场景、签名使用的私钥保存、私钥丢失能否保护用户财产。

4) 密码算法。密码算法主要测试 4 个指标：密码算法种类、国密算法、加密机、可插拔自定义密码算法。

5) 匿名性。匿名性主要测试 5 个指标：交易匿名、全部匿名还是部分匿名、子账户对其父账户保持交易透明、子账户对其他父账户保持交易匿名、对审计或超级权限账户保持交易透明。

6) 抗攻击能力。抗攻击能力主要测试 4 个指标：抵御 DDoS、黑客等网络攻击、抗攻击能力程度、最大容忍节点失效个数、最大容忍节点欺骗个数。

## 11.3 挖矿和共识协议的弱点

### 11.3.1 中本聪一失之虑

比特币的设计者中本聪毕竟不是神，他在设计比特币时，以去中心化为根本前提，原本希望利用分散的计算资源，通过每个人的计算机解决算力问题，以完成区块链共识和交易上所必需的复杂计算，所以设计了 PoW，以奖励参与计算的节点。在这份协议的基础上，全球每个比特币的参与者都可以开动自己的电脑进行挖矿，公平参与来获取比特币。但后来的发展应该是设计者不想看到的：由于 CPU 挖矿效率极低，人们发现 GPU 效率相对高点，发展出 GPU 挖矿，再后来有 FPGA，最后发展到 ASIC（专用集成电路）挖矿，如果说 GPU 挖矿还和“人人为我我为人人”沾点边的话，那么 ASIC 专业矿机的诞生就彻底背离了当初中本聪“一机一票”的设计初衷。

从此 PoW 机制仿佛进入了军备竞赛，最后矿机的集中分布形成了矿场，算力的中心化让后来者无法公平参与。同时算力的过于集中不利于整个比特币网络的安全性，毕竟还有著名的“51% 攻击”的潜在威胁。

所谓“矿池”，就是大家整合算力，一起解决同一道题。这种方式需要统一管理，一般做法是矿池开出任务，并准备适合的奖励措施，无论算力高低，把一种类似赌运气的挖矿赌博变成一种统计学的回报，以降低回报的波动性。这种工业式的挖矿革命将随着 PoW 共识机制的推广而逐渐完善。

### 11.3.2 挖矿和算力集中困境

矿池算力集中化带来的相关安全话题是一个热门话题。曾经有一段时间，比特币是一个大众参与的游戏，人们使用自家的家用电脑可以碰运气挖比特币。然而，随着挖矿难度的增加，专业的计算机芯片和专业的比特币采矿集团出现了，并形成了一些大规模的矿池。它们主要分布在中国，以及一些电力便宜的国家 and 地区。比起区块链技术应用落地的困境，比特币“挖矿”的生意风景这边独好。这里展开的，是一场“算力”的军备竞赛。由于比



特币的机制，矿圈的这场“权力游戏”，胜者甚至可以决定比特币的命运和走向。

因为大多数区块链共识算法的“51%攻击”问题，以及日益集中的矿池算力，导致我们的争议焦点集中在去中心化系统以及实际上日益集中的算力导致的中心化争论上。目前实际上并未出现51%攻击的问题，但不可否认，算力的集中至少违背了比特币去中心化的初衷，成为其继续发展的一大隐患。

### 11.3.3 其他共识算法及其问题

区块链的自信任主要体现在分布于区块链中的用户无须信任交易的另一方，也无须信任一个中心化的机构，只需要信任区块链协议下的软件系统即可实现交易。

这种自信任的前提是区块链的共识机制，即在一个互不信任的市场中，要想使各节点达成一致的充分必要条件是每个节点出于对自身利益最大化的考虑，都会自发、诚实地遵守协议中预先设定的规则，判断每一笔记录的真实性，最终将判断为真的记录记入区块链中。换句话说，如果各节点具有各自独立的利益并互相竞争，则这些节点几乎不可能合谋欺骗用户。而当节点们在网络中拥有公共信誉时，这一点体现得尤为明显。

表 11-1 所示为区块链各种共识机制的比较。

表 11-1 区块链各种共识机制的比较

| 共识机制 | 代表                         | 优点                          | 缺点                        |
|------|----------------------------|-----------------------------|---------------------------|
| PoW  | 比特币，莱特币                    | 实现简单<br>安全可靠<br>网络资源消耗小     | 计算资源消耗大<br>易分叉<br>速度慢     |
| PoS  | Peercoin, NXT<br>和 Qtum    | 资源消耗少                       | 实现复杂<br>安全漏洞<br>网络资源消耗大   |
| DPoS | Bitshares, EOS,<br>LISK    | 资源及网络资源消耗小<br>吞吐量高<br>共识时间短 | 实现复杂<br>安全漏洞              |
| PBFT | Fabric0.6                  | 共识效率高，可实现高频交易               | 当系统只剩下 33% 的节点运行时，系统会停止运行 |
| DAG  | IOTA, Byteball,<br>Nerthus | 高吞吐量，高并发<br>异步通信            | 实现复杂<br>同步难<br>网络资源消耗大    |

共识机制最好的设计是提供可插拔模块化共识。共识算法的选择与应用场景高度相关，不同的应用应该有不同的共识算法的选择。

## 11.4 交易效率问题

### 11.4.1 比特币和以太坊的交易效率困境

谈到交易效率，首先我们需要一个衡量指标，在区块链系统中用 TPS 指标来衡量一个

系统的交易效率。

TPS 可基于测试周期内完成的事务数量计算得出。例如，用户每分钟执行 6 个事务，TPS 为  $6 \div 60s = 0.10$  TPS。同时我们会知道事务的响应时间（或节拍），如此例中，60 秒完成 6 个事务也同时代表每个事务的响应时间或节拍为 10 秒。

一个系统吞吐量通常由 TPS、并发数 2 个因素决定。每套系统的这两个值都有一个相对极限值。在应用场景访问压力下，只要某一项达到系统最高值，系统的吞吐量就上不去了，如果压力继续增大，系统的吞吐量反而会下降，原因是系统超负荷工作，上下文切换、内存等等其他消耗导致系统性能下降。

根据这个指标，以比特币为例，其 TPS 约为 7，以太坊大概为 20，换算到一天 24 小时的概念，大概相当于比特币 30 万笔/天，以太坊 45 万笔/天。

根据 Blockchain 的数据，投资者大概需要平均 78 分钟来确认一次比特币交易。个别时段，这一平均时长一度高达 1188 分钟，也就是将近 20 小时。这将大大降低用户使用比特币的兴趣。

### 11.4.2 比特币扩容

一种货币不是天生就有价值，而是很多人相信它有价值的时候才会有价值。这也可以说在一定群体里形成的“共识机制”。由于前面提到的比特币交易速度慢的问题，比特币扩容一直是社区里被广泛重视的一个重要问题。

首先要澄清的一点是，比特币扩容不是增发比特币，而是针对比特币交易量不足增加比特币的交易量上限。而为什么要增加交易量上限呢？

比特币最初的设计是规定每 10 分钟（左右）挖出一个大小为 1MB 的区块，每笔交易平均下来是 250 字节，于是，每块可以放进 4000 笔交易，换算成每秒则近似等于每秒 7 笔交易。这个数字太小了，比如 Paypal 是每秒 100 笔量级的，而像支付宝这种集中式交易系统，在“双十一”的时候可是每秒 100 000 笔量级的。和它们一比，比特币根本就不能称之为交易系统。

在比特币的扩容问题上有两个技术难题。一是技术上如何实现，即扩容多少合适。第二个问题是如何实施。考虑到比特币系统是一个分布式系统，它面临的实施层面的问题会更大，因为涉及在实施层面上的新旧节点如何协调，新旧账本如何共存和互认的问题。

分布式系统的升级必须让每个节点都升级才行。如果有人不升级，将会产生分叉。关于分叉，前文详细讲解过，此处不再赘述。

### 11.4.3 比特币的隔离验证、闪电网络与侧链

区块链的可扩展性问题，至今仍旧是一个学术难题。例如交易量问题一直是比特币的最大问题，那么如何实现比特币交易量扩容？简单说应该有以下几种方式。



1) 改变比特币采用的 PoW (工作量证明) 共识机制。这相当于改变了比特币的安全机制, 或者说采用其他共识机制来进行比特币的升级很难得到原有比特币相关人员的认同。因为比特币已经承载了太多的用户和价值, 甚至还承担了整个数字货币和区块链旗帜的重任。想要完全更改共识算法, 冒着不可预测的风险去扩容, 无论是投资者、矿工、开发者, 都不会轻易答应的。

2) 可以改变区块大小, 但是不能增得太大, 因为数据存储和传输工作量会大大增加。

3) 改变区块生成间隔。这个间隔可以缩小, 但是会使得比特币的网络出现大量的孤块和分叉, 造成大量的算力浪费和安全隐患。

4) 修改数据存储规则, 相同大小的区块能承载更多交易。

隔离验证就是一种能够在不增加区块大小的前提下增加交易容量的方法。

前文介绍过, 比特币的一笔交易大概 250 字节, 主要数据包括以下 2 个部分:

1) 转账记录, 也就是交易方和交易额;

2) 用户有权利做这笔交易的证明, 这个证明是一组数字签名。

实际上, 比特币的签名数据很大。粗略估计, 签名大概占了交易大小的  $2/3$ 。

这样, 我们就可以把交易里所有的签名单拉出来, 然后把所有的签名打包放在数据块的后面, 可以节省大约  $2/3$  的空间。也就是说, 每笔交易都被分成了 2 部分: 交易和见证(签名)。交易部分只有 100 字节左右, 于是一个 1MB 的区块里面能放 10 000 笔交易(原来是 4000 笔), 然后, 所有的见证部分, 大约 1.5MB ~ 2MB, 都被放到了后面。采用了隔离验证技术的新节点当然可以接受这种数据格式, 而旧节点虽然不能识别数据块后面的见证部分, 但它们仍旧会认为前面的部分是合法的区块。这样通过软分叉就可实现隔离见证的升级。采用这种隔离验证的方法, 大约可以让比特币提高 2 ~ 3 倍的交易量。

除此之外, 隔离验证还修复了比特币一个被称为“可变性”(Malleability)的缺陷。比特币的签名方式比较复杂, 其签名只针对 UTXO, 并不包含交易里的所有信息。这样的话, 攻击者可以改变交易里的信息, 也可以改变交易 ID, 但签名信息仍然有效。隔离见证将签名信息从交易中提出来, 可以针对整个交易签名, 使得比特币的交易不能改变, 交易 ID 可以固定。更重要的是, 可以更容易地实现一个叫作“闪电网络”的技术。

闪电网络是比特币的一个不改变主链结构的扩容机制, 简单说就是一个在链上提供担保的链下交易机制。实际上就是在比特币上签署一个协议, 在比特币主链以外再架设一个通道, 用户的币存在这个通道上可以进行快速支付。闪电网络是一个去中心化架构的网络, 和传统的交易所有本质区别。

例如你经常给某个人转钱, 你不用每次都把交易上传到比特币的链上。你们双方可以先在比特币上签署一个协议, 交一笔保证金, 然后只要你们之间转账的总额不超过保证金, 转账就可以私下进行, 而这个协议保证如果对方耍赖了, 你可以凭着转账记录上传到区块链上把属于你的钱拿走, 不用再次通过对方授权。这种方式其实和主链交易没有可比性, 它完全是另一种交易形式, 提供了另外一种支付手段。闪电网络可以将一部分交易挪到链

下进行，减轻了主链的负担。

在交易过程中，闪电网络允许创建“微支付渠道”，类似交易双方建立一个交易链，多笔比特币交易在无须与主链进行互动的情况下，仍能安全地进行。这些在通道中的支付交易速度极快，与当前的比特币支付需要冗长的交易验证时间不同，交易链中的交易只有最后一笔需要真实地进入比特币区块链。如果任何一方终止合作，或者说在约定的时间内没有响应，该通道可以被关闭。重要的是，这种支付是可路由的，它是跨越多跳路径建立的专门通信管道。相较于为每一个新的合约方创建一个渠道，你可以维持一些渠道，连接少数良好的安全中介机构，并通过他们来完成交易。这就是简单的支付通道背后的思路。目前已经存在这种支付通道。你可以一直向某人发送可替换的交易，每次额度比上一次大一点点，一旦达到某种条件，通道被终结，只有最后一笔支付向全网广播。

事实证明，只需要少量几乎没有争议的比特币升级，人们就可以生成更加通用的支付通道，它允许双向支付，也允许“条件支付”。条件支付允许用户构建一个支付网络。实际上，用户可以通过安全和非信任依赖的方式设定一些条件，例如“如果张三支付了李四，我就支付给张三”。一些事情发生之后，用户的钱包就会自动向比特币网络广播这个条件支付交易，然后等待。

从理论上讲，这种分布式小额支付网络（闪电网络）可以将比特币的日交易量扩充到数十亿笔，并且极少地使用到区块链，以及仅需少量的交易费。

当然这种闪电网络的机制也存在安全漏洞，如果在这个通道里的交易在未得到主链的确认之前出现安全问题，将会导致一些交易无法确认，造成财产损失。

闪电网络论文提出了生成通道和支付网络的机制。它也是如今比特币创新的一个热点。

侧链是以锚定比特币为基础的新型区块链，就像美金锚定到金条一样。比特币在区块链里相当于货币体系的黄金地位，具有最去中心化、最多分布节点、最公平区块链。侧链是以融合的方式实现加密货币金融生态的目标，而不是像其他加密货币一样排斥现有的系统。利用侧链，我们可以轻松地建立各种智能化的金融合约，如股票、期货、衍生品等。用户可以有成千上万个锚定到比特币上的侧链，特性和目的各不相同，所有这些侧链依赖于比特币主区块链保障的弹性和稀缺性。比较著名的比特币侧链是 Rootstock 和 BlockStream 推出的元素链。

#### 11.4.4 基于 DAG 的提速技术

DAG (Directed acyclic graph)，有向无环图，是计算机领域一个常用的数据结构。因为独特的拓扑结构所带来的一些特性，它经常被用到处理动态规划、导航中寻求最短路径、数据压缩等场景中。

第一次提出 DAG 跟区块链结合是在 Nxt 社区，可以发现 DAG 最初出现就是为了解决区块链的效率问题。比特币的效率一直比较低，基于 PoW 共识下的出块机制是一个原因，由于链式的存储结构，整个网络中同时只能有一条链，导致出块无法并发执行。社区有人



提出用 DAG 的拓扑结构来存储区块,这个时候更多还是类似侧链的解决思路,不同的链条存储不同类型的交易,降低出现“双花”的可能,在之后某个节点需要合并的时候,几个分支再归并到一个区块。

通过以上设想我们可以改变区块的链式存储结构,变成区块 DAG。在区块打包时间不变的情况下,网络中可以并行地打包  $n$  个区块,网络中的交易就可以容纳  $n$  倍。

这时候 DAG 跟区块链的结合还是停留在侧链的思路,不同类型的交易可以并行在不同的链条进行,达到提升性能的目的。这时候的 DAG 还是有区块的概念。

但是区块的概念其实也是妨碍我们提升效率的关键因素。那么,可不可以没有区块的概念呢?为什么一定需要区块呢?能否让每一笔交易直接参与维护全网的交易顺序?这样交易被发起后直接跳过打包区块的阶段,直接融入全网,如此达到所谓的“无区块”(blockless)效果。这样确实连打包交易出块的时间都省去了,DAG 最初跟区块链的结合就是为了解决效率问题,现在不用打包确认,交易发起后直接进入确认网络,理论上效率自然提高很多。

自此,以 blockless 独树一帜的 DAG 区块链雏形基本形成。其中 IOTA 和 Byteball 在市场上的表现最为耀眼。

DAG 系的区块链有些概念很有趣,了解这些概念更容易理解 DAG 技术。

DAG 与链式结构的本质区别在于异步与同步通信。DAG 通过将事务操作进行异步处理来增加网络吞吐量,采用某种传播算法在节点间发送操作日志,并通过某种机制(IOTA 每次验证前 2 条交易,并计算一个 PoW 代表权重)将一个权重赋给该操作。相比起同步操作的链式结构,DAG 结构与任何异步机制一样,能够带来的提升在于吞吐量,但是产生的问题则在于无法有效预测交易被确认的时间与周期。

DAG 网络一个重要的问题就是解决网络宽度。DAG 网络中,每笔交易被确认,都需要链接到已经在网络中存在的并且比较新的交易;如果都选择网络中比较早的交易,会导致网络宽度过宽,新的交易难以得到确认。理想的状态是,新的交易发起时,选择网络中已经存在并且比较新的交易做链接确认,这样网络的宽度保持在一定范围,能让新的交易有足够快的确认时间。

DAG 的主要特点如下。

1) 交易速度快: DAG 摒弃了区块概念,交易直接进入全网中,所以交易速度预期比基于 PoW 和 PoS 的需要出块的区块链会快很多。

2) 无须挖矿: DAG 把交易确认的环境直接下放给交易本身,无须由矿工打包成区块后同意交易顺序。所以 DAG 网络中没有矿工的角色。

3) 无手续费: 交易发起只需要做简单的 PoW 工作量证明,整个网络中的 PoW 都是发起交易者自己做的,而不是交给矿工,所以发起交易无须手续费。

4) 需要见证节点: DAG 需要见证人机制的存在。这一部分不管是 DPoS、PoS、PBFT,大家最终都会在效率、安全性上寻求一种平衡。

### 11.4.5 其他提速思路

目前区块链效率问题比较突出，但是笔者相信随着时间的推移会有更多的新技术产生，提速可以从以下几个方面来考虑。

- ❑ 网络带宽：网络带宽的发展会进一步允许更好更先进的分布式共识机制的产生。
- ❑ 硬件速度：包括各种 CPU、GPU 等硬件速度的不断提高会大大提高区块链的效率。
- ❑ 共识算法：目前的算法都有缺陷和不足，将逐步发展，协议也将不断完善。
- ❑ 并发执行：链状结构理论上缺乏并发机制，类似 DAG 之类的异步并发技术也将不断完善。
- ❑ 数据格式：区块链里存储的数据格式可通过压缩技术存储，提高区块链的运行效率。

## 11.5 系统升级维护问题

### 11.5.1 硬分叉史记

以太坊：2017 年成功实施的“拜占庭”硬分叉，通过对底层协议的升级创建新的规则，来完成对整个系统的性能提升和功能增设。2017 年 10 月 16 日，以太坊官方宣布，“拜占庭”在以太坊第 437 万个区块高度成功实施硬分叉，将协议升级后调整了区块奖励，减小区块大小，并为平台上包括区块链隐私性等问题引入了零知识证明等解决方案，完成了开发人员打造更为适合去中心化应用生态环境的技术目标。

#### （1）以太坊（ETH）和以太经典（ETC）

TheDAO 计划基于以太坊智能合约建立一个众筹平台，于 2016 年 5 月正式发布，截至当年 6 月，募集资金超过 1.6 亿美元。之后，TheDAO 被黑客利用智能合约的漏洞，转移了市值五千万美元的以太币。为了挽回投资者资产，以太坊社区投票表决决定将更改以太坊代码，希望索回资金。为此，以太坊在第 1 920 000 区块进行硬分叉，回滚所有以太币（包括被黑客占有的）。

但是，有一部分人认为以太坊这种做法违背了区块链的去中心化和不可篡改精神，坚持在原链上挖矿，从而形成两条链：一条为不承认回滚交易的链——以太经典（ETC），一条为承认回滚交易的链——以太坊（ETH），各自代表不同的社区共识以及价值观。分叉时持有以太币的人在分叉后会同时持有 ETH 和 ETC。

以太雾（ETF）是以太坊在 2017 年 12 月 14 日分出的公有链，主打雾计算的创新技术，增加了分布式储存和分布式计算的能力，拥有缩短网络延迟、节省计算资源、减小核心网络压力的优势，同时带来更高的可靠性。

#### （2）比特币黄金

比特币黄金（Bitcoin Gold，BTG）是在 2017 年 10 月 25 日左右的又一次比特币硬分叉产物。比特币黄金是对大矿工们集中算力的一种对抗，比特币矿工越来越多地使用定制的 ASIC 来挖矿，但 ASIC 价格昂贵，只有大量买进时才会降低价格，所以大矿工们可以通过



较低价大量买进 ASIC 来获得巨大的算力, 导致比特币网络的算力被集中。比特币黄金将使用一种新算法来避免情况恶化。

### 11.5.2 系统升级维护难题和分叉

区块链系统的升级维护包括硬件和软件两个方面。

硬件方面将随着技术的发展而逐渐完善, 随着区块链的发展, 将有更多的区块链硬件设备涌现, 包括矿机、芯片、区块链硬件节点, 以及各种应用相关类硬件出现。比如基于区块链系统设计的各种物联网设备、智能电表、水表、气表、手机以及各种穿戴设备等。

软件方面的升级主要包括以下几个方面。

1) 数据结构: 例如比特币现金 (BCH) 从 1MB 区块改为 8MB 区块就是一种数据结构的升级。

2) 共识算法: 随着系统的运行, 最初设计的共识算法满足不了系统日益发展的需求时可以修改共识算法, 带来的将是系统的升级。

3) 加密算法: 算法问题是区块链安全的核心, 再安全的算法也是百密一疏, 因此随着黑客技术的进步将不断催生加密算法的改进和升级。

4) 应用升级: 系统的应用发生改变时也面临升级问题。

5) 漏洞修补: 任何软件系统将不断完善, 寻找自身漏洞并不断完善。

以上任何一种类型的修改都将可能导致分叉的出现。当然分叉也包括人为因素, 因为毕竟区块链网络是由人来控制的。

分叉后不可避免地会出现新、旧链之间的数据协调以及用户的分流问题。

## 11.6 小结

本章介绍了许多区块链领域内的常见问题, 这些问题估计会困扰区块链的初学者。本章基本内容包括以下几个方面: 区块链的技术局限、数据冗余、安全性、挖矿、各种共识协议的弱点、交易速度、安全问题、51% 攻击问题、女巫攻击、交易所、以太坊智能合约安全漏洞、挖矿和算力集中困境, 比特币和以太坊的交易效率困境、比特币的扩容、隔离验证、闪电网络、侧链、DAG 技术、硬分叉历史、系统升级维护等问题, 涵盖了区块链各有可能出现问题的领域。

### 习题

- [1] 区块链的安全包括哪几个方面?
- [2] 女巫攻击包括哪些类型?
- [3] 比特币 51% 攻击的原理是什么?
- [4] 简述 DAG 的基本概念。
- [5] 请给出 TPS 的定义及计算公式。

## 区块链评测

作者：刘胜

区块链是一种战略性、颠覆性的新兴技术架构。世界各国政府、大型金融机构、企业集团以及众多初创公司都纷纷投入了大量资源对区块链进行研究和开发。与此同时，各个区块链联盟、区块链协会以及各大校企合作的区块链实验室也纷纷成立。各种区块链层出不穷，但鱼龙混杂。

所有区块链行业的玩家都希望率先成为制定标准的一方，以争夺区块链标准的话语权，占领行业制高点，并从中抢占某些先机。

如图 12-1 所示，本章将系统地介绍区块链评测的难度、策略、各层的评测点，以及相关辅助工具。

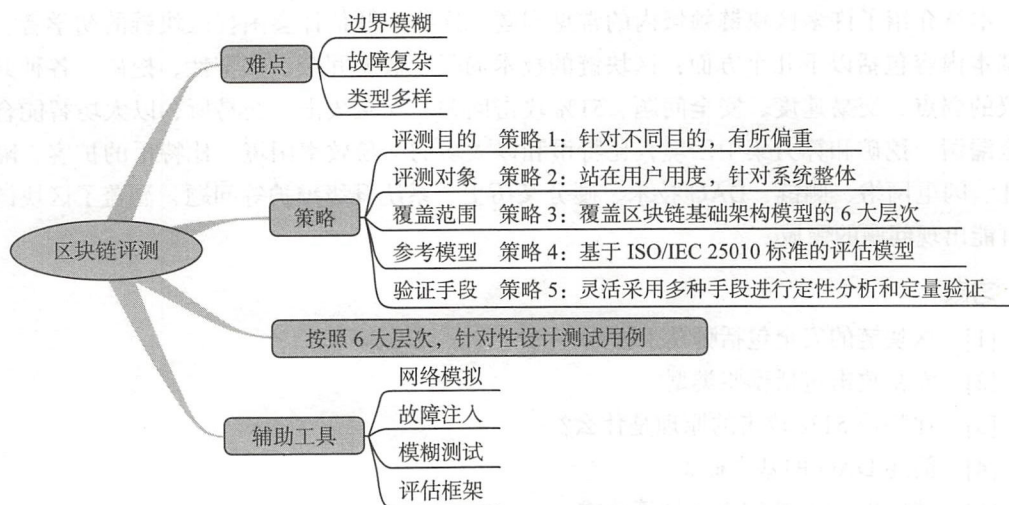


图 12-1 区块链评测的要点



## 12.1 评测的难点

与常规的集中式系统相比，分布式系统要复杂得多。实现一个分布式系统很难，做好分布式系统的测试同样不简单。而区块链作为一类“无中心”网络系统，比起常规的分布式系统还有“去中心”的特点。因而对区块链的测试和评估也比较困难。

区块链系统的测试与传统的软件测试有很大的不同。

### (1) 难点之一：系统边界模糊（图 12-2）

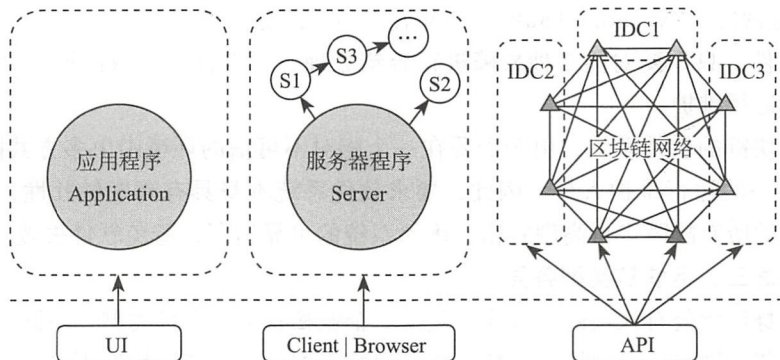


图 12-2 区块链测试难点之一：系统边界模糊

传统的软件，不管是独立的应用程序还是 C/S 模式、B/S 模式的服务器程序，都有明显的中心节点和围绕中心节点的系统边界。前者可以通过用户界面进行测试，后者可以通过专用客户端或通用浏览器，通过固定的网络接口进行测试。

而区块链系统，则是一个多中心或无中心、更加分布式的网络。这个网络有可能跨越多个子网、多个数据中心、多个机构、多个运营商、甚至多个国家，其边界是模糊的。对于区块链底层的测试，不仅仅是前端 API 与某个区块链节点之间的测试，还得考虑大量区块链节点之间的交互。

### (2) 难点之二：软件故障类型复杂

由于区块链是一种多方共同维护的 IT 系统，需要尽可能减小软件失效，提高软件可靠性。按照软件可靠性模型，软件失效（Software Failure）的机理如图 12-3 所示。

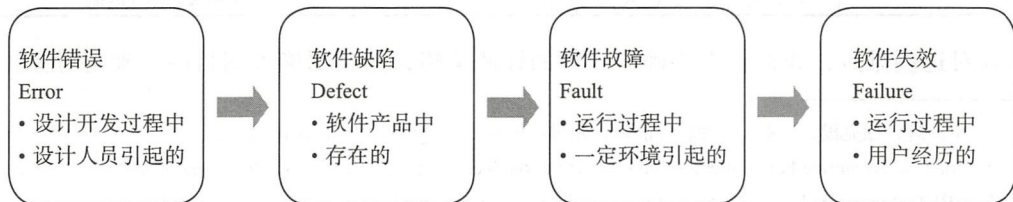


图 12-3 软件失效的机理示意图<sup>①</sup>

① 吴玉美，阮镡. 软件可靠性测试的加速机理研究.《计算机应用》，2006 年 6 月。

其中,软件故障(Software Fault)是指软件运行过程中出现的一种不希望或不可接受的内部状态。此时,若无适当的措施(容错或异常处理机制)加以及时处理,便会产生软件失效<sup>①</sup>。

虽然软件错误、软件缺陷、软件故障都有可能導致软件失效。但对于区块链系统而言,由于其运行环境的复杂性,因此需要尤为重视软件故障的异常和容错处理。

一般的软件故障有3类:

- 1) 宕机故障(Crash Fault),外部表现为“服务不再可用”;
- 2) 宕机-恢复故障(Crash-Recovery Fault),外部表现为“丢消息”;
- 3) 拜占庭故障(Byzantine Fault),外部表现为“部分消息不可信”。

传统的软件,只需要对前2种故障进行容错处理,当软件由于故障而失效时,可以由相关人员进行运行维护。

而基于区块链的软件系统,由于需要在一个相对不可信的环境中由多方共同运行维护,一般未配备统一的运行维护人员。因此,要求软件系统本身具有高度健壮性,能对包括拜占庭故障在内的所有故障进行高度容错,确保系统的可靠运行,避免软件失效。

### (3) 难点之三:区块链类型各异

区块链本身包含公有区块链、行业联盟链、企业私有链等多种类型,不同类型的区块链在治理模式、节点许可性、网络覆盖性、节点规模等方面,其特征均有不同。而区块链评测需要考虑支持全部的类型,导致测试方案更加复杂。表12-1为不同类型区块链的主要差异。

表 12-1 不同类型区块链的主要差异

| 区块链类型  | 公有区块链  | 行业联盟链            | 企业私有链  |
|--------|--|------------------|--|
| 治理模式   | 社区共同治理   | 多方共同维护和治理        | 单方维护   |
| 经济激励   | 必须   | 可选               | 无  |
| 节点许可性  | 无须许可   | 需要许可             | 需要许可   |
| 网络覆盖   | 广域网  | 广域网              | 局域网、专网                                       |
| 典型节点规模 | 很大(> 1000)                                     | 较大(100 ~ 1000)   | 较小(< 100)                                    |
| 共识确认方式 | 多次确认   | 立即确认             | 立即确认   |
| 一致性    | 大概率一致性   | 确定性一致性           | 确定性一致性                                       |
| 常用共识算法 | PoW、PoS、DPoS等                                  | Paxos、PBFT、Raft等 | Paxos、PBFT <sup>②</sup> 、RBFT <sup>③</sup> 等 |
| 交易处理时间 | 长(如n分钟)  | 短(如n秒)           | 短(如n秒)                                       |
| 典型应用场景 | 加密数字货币<br>如:比特币 <sup>④</sup> 、以太坊 <sup>⑤</sup> | 支付清算网络           | 私有账本系统<br>如超级账本 Fabric <sup>⑥</sup>          |

针对这些难点,我们首先要确立合理的评测策略,包括明确评测目的、确定对象、界

① 单锦辉,徐克俊,王戟.一种软件故障诊断过程框架.《计算机学报》,2011年2月。

② Practical Byzantine Fault Tolerance and Proactive Recovery. <http://pmg.csail.mit.edu/papers/osdi99.pdf>, 1999.

③ RBFT: Redundant Byzantine Fault Tolerance. <http://www.pakupaku.me/plaublicin/rbft/report.pdf>, 2015.

④ Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. <http://bitcoin.org/bitcoin.pdf>, 2008.

⑤ 参见 <https://github.com/ethereum/wiki>.

⑥ 参见 <https://www.hyperledger.org/projects/fabric>.



定评测范围、选择参考模型、确定评测内容、选择评测手段等。

## 12.2 评测的策略

### 1. 评测目的

根据评测目的不同，区块链评测服务可分为以下几种。

- ❑ **软件产品登记测试**——是进行软件产品登记、双软认证（软件企业认证和软件产品认证）的必备测试。测试内容包括安装与卸载、功能、安全性、本地化、用户文档和病毒检查等。
- ❑ **软件成果鉴定测试**——依据其科研项目计划任务书或技术合同书，参照相应的国家标准对要申请鉴定的软件类科技成果进行的一种符合性测试，以验证其是否符合它的规格说明、是否达到预定的计划目标。其测试结果可作为软件类科技成果鉴定的依据，如创新基金申请、高新技术转化测试、成果鉴定、创新基金验收等。
- ❑ **软件比对与选型测试**——根据客户需求，对指定的同类型软件进行比对测试。比对内容包括功能、性能、可靠性、安全性、易用性、兼容性及其他技术指标等，为客户的软件采购、软件选型、市场推广提供客观依据和建议。
- ❑ **区块链应用验收测试**——也称为交付测试，是软件开发结束后，对软件产品投入实际应用以前进行的最后一次质量检验活动。确保软件准备就绪，其功能和性能等方面符合用户的合理预期。
- ❑ **区块链应用系统测试**——包括区块链应用产品评估和区块链应用系统评测。前者在关注功能测试、性能等技术指标测试的同时，还要求其符合产品标准或企业标准。后者与常规软件测试类似，包括了功能、性能、安全等方面的测试。

针对不同评测目的，其评测内容也有不同的侧重点：

- ❑ 软件产品登记测试，更偏重于功能测试；
- ❑ 软件成果鉴定测试，更偏重于一些技术指标检测，说明其技术创新性或领先性；
- ❑ 软件比对与选型测试，需要从多个候选产品之间，提取相同评测点，进行比对；
- ❑ 区块链应用验收测试较为基础，主要确保功能、性能等方面符合用户预期；
- ❑ 区块链应用系统测试相对全面，涵盖功能性测试、非功能性测试等多方面。

本章将针对多种评测目的，结合区块链特点，通过预设的使用场景，尽可能多地覆盖所有测试项。

### 2. 评测对象

由于区块链系统通常涉及多方机构、跨越复杂网络，其物理边界尤其模糊。此外，区块链通常是一种“永不停机的低成本系统”<sup>①</sup>，其时间边界也很模糊。因此，针对区块链系

① <https://www.ccn.com/japan-evaluation-blockchain-platforms/> Japanese Government Unveils Evaluation Process for Blockchain Platforms.

统的评测, 需要尽可能地界定其评测对象和范围, 否则难以实施。

本评测对象及范围如图 12-4 所示。

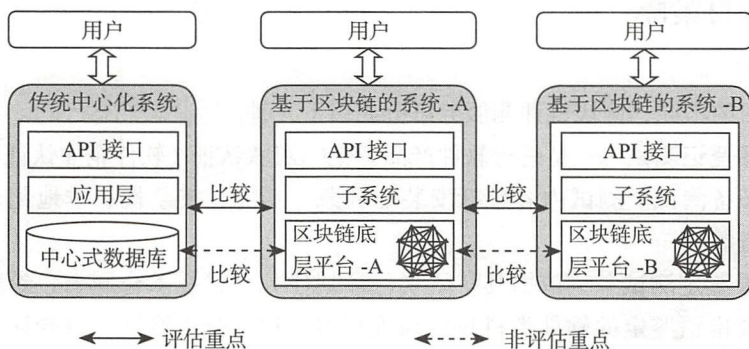


图 12-4 评测对象及范围示意图

简单来说, 本评测的对象是“基于区块链的系统”(Blockchain-based System), 包括区块链底层平台、基于底层平台的核心子系统以及相关 API 接口服务。

一般而言, 站在用户的角度对“系统整体”进行功能和性能评估测试, 比仅针对“区块链底层平台”来评测更加重要。

### 3. 覆盖范围

本评测的范围将覆盖图 12-5 所示的区块链技术典型基础架构模型所涉及的 6 大层面。

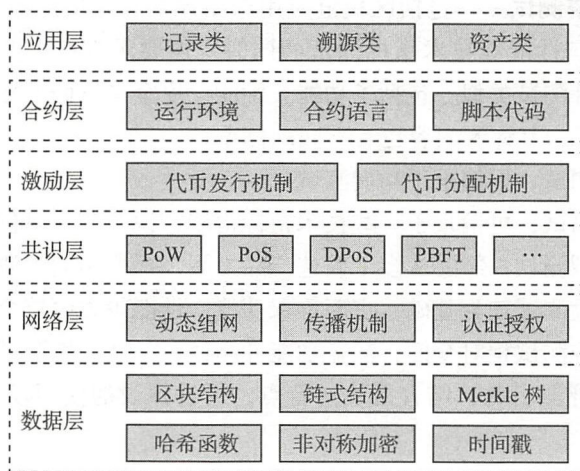


图 12-5 区块链基础架构模型<sup>①</sup>

各个层面评测内容概述如下。

□ 数据层: 通过某种特定数据结构的“区块”构成“链”式结构, 并综合使用一系列

① 袁勇, 王飞跃. 区块链技术发展现状与展望 [J]. 自动化学报, 2016 年 4 月。



密码学相关技术，一方面确保数据的安全可信，另一方面实现数据的隐私保护。

- ❑ 网络层：包括了节点管理，多个节点之间的认证方式、授权方式、组网方式，以及是否支持在任何时候新节点的加入和旧节点的退出。
- ❑ 共识层：主要封装网络节点的各类共识算法，实现分布式场景下的一致性和正确性。对于评测而言，需要分析其共识机制的原理，是否支持拜占庭容错，能支持多大的节点规模。
- ❑ 激励层：是否支持类似比特币、以太坊之类的“原生代币”，从而可以将一些经济机制集成到区块链技术体系中来，主要包括经济激励的发行机制和分配机制等。
- ❑ 合约层：对“智能合约”的支持程度，包括可编程脚本的运行环境、合约语言、脚本代码。可以是类似比特币系统中非图灵完备的脚本，也可以是类似以太坊中支持图灵完备的自定义编程语言 Solidity；或者是类似超级账本 Fabric 中直接使用支持图灵完备的主流编程语言 Golang。
- ❑ 应用层：封装了各种场景的具体区块链应用。从应用属性来分，可以分为存证、溯源、资产 3 大类应用。对于不同类应用，其记录的内容以及相关的操作都会有所不同，需要区别对待。

#### 4. 参考模型

基于区块链的系统对软件质量的要求很高。一方面，区块链底层本质上是一个分布式账本的“框架”，其功能特性相对简单，但是其非功能性比较复杂。另一方面，基于区块链的系统，其软件生命周期远长于传统软件。例如比特币系统已经运行了 9 年多，未来还将运行几十年，甚至上百年。因此，传统的功能测试、性能测试已经不能满足区块链测试的需求。

这里，我们将参考“ISO/IEC 25010：系统和软件质量需求和评估（SQuaRE）”标准来设计区块链的测试、验证点，如图 12-6 所示。

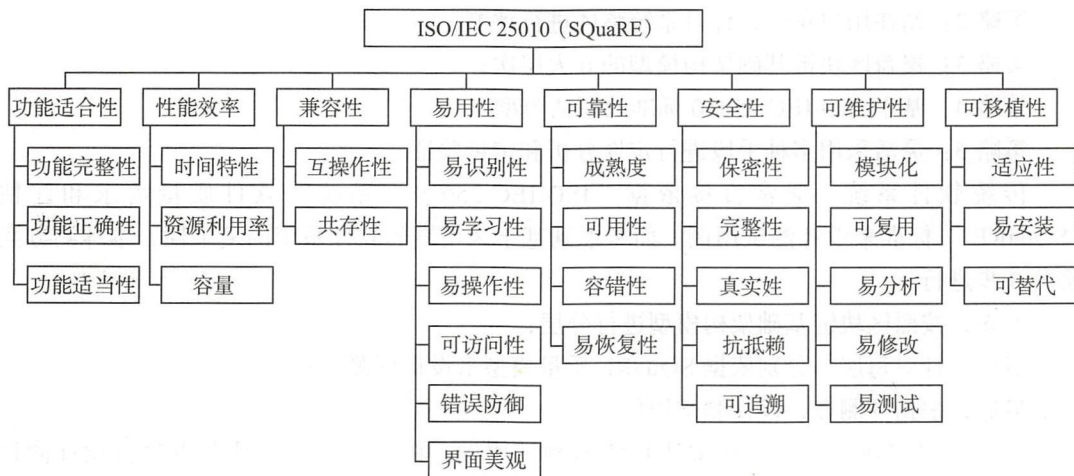


图 12-6 系统和软件质量需求和评估模型 (ISO/IEC 25010)

如图 12-6 所示, ISO/IEC 25010 标准模型分为 8 个维度。

- 功能性测试, 包括功能完整性、正确性、适当性;
- 性能效率测试, 包括时间特性、资源利用率、容量;
- 兼容性测试, 包括互操作性、共存性;
- 易用性测试, 包括易识别性、易学习型、易操作性、可访问性、错误防御、解密美观;
- 可靠性测试, 包括成熟度、可用性、容错性、易恢复性;
- 安全性测试, 包括保密性、完整性、真实性、抗抵赖、可追溯;
- 可维护性测试, 包括模块化、可复用、易分析、易修改、易测试;
- 可移植性测试, 包括适应性、易安装、可替代。

本评测方案将针对各个层面的评测点, 依据这 8 个维度去验证。

### 5. 验证手段

参考软件成熟度评测方法, 将区块链软件成熟度评测方法分为 4 类。

- 手工测试验证: 按照用户使用手册, 模拟用户正常使用来验证其功能。
- 脚本测试验证: 借助现有工具或编写脚本程序进行测试, 实现定量分析。
- 工具测试验证: 使用专用基准测试工具, 给出具体的评测数据和期望结果。
- 专家判断评估: 对无法通过上述方法完成的测试项, 通过专家判断进行评估。

其中, 手工测试和专家判断主要实现定性分析, 而脚本测试和工具测试主要实现定量分析。在后续针对 6 大层面而设计的测试用例中, 可以灵活运用定性和定量方法, 对各个评测点进行验证。

### 6. 小结

总体来说, 本文的区块链评测方案, 将采用如下一些策略。

- 策略 1: 针对不同评测目的, 测试用例有所偏重;
- 策略 2: 站在用户角度, 针对系统整体进行评测;
- 策略 3: 覆盖区块链基础架构模型的 6 大层次;
- 策略 4: 基于 ISO/IEC 25010 标准的评估模型;
- 策略 5: 灵活采用多种手段进行定性分析和定量验证。

传统软件系统, 通常直接依据“ISO/IEC 25010: 系统和软件质量需求和评估 (SQuaRE)”标准来设计测试用例。而考虑到基于区块链的软件系统的复杂性, 本评测方案将分 3 步进行:

- 首先, 按照区块链基础架构模型进行分层;
- 然后, 针对每层, 分别依据 SQuaRE 质量模型来设计评测点;
- 最后, 根据评测点, 设计测试用例。

本章主要讲述评测点, 而细述具体的测试用例。读者可以根据评测点自行设计测试用例。



## 12.3 应用层评测

如图 12-7 所示，区块链应用系统按产品属性可以分为存证、溯源、资产 3 大类。

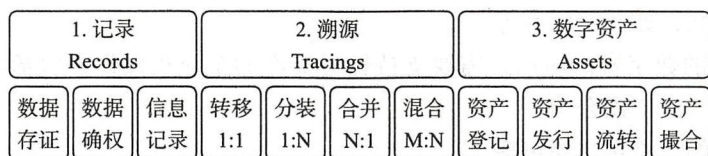


图 12-7 区块链应用的产品属性分类

### 1. 存证类应用

存证类应用主要用做信息记录。

- ❑ 数据存证 (Proof of Existence): 只需保存数据的哈希指纹, 证明数据在某个时间段存在。
- ❑ 数据确权 (Proof of Ownership): 除了保存哈希指纹外, 还需保持数字签名信息, 以证明某项数据在某个时间点存在, 而且归属于某个主体。
- ❑ 信息记录则需要记录更多的数据信息, 将区块链当成一个高度防篡改的文档型数据库。

存证类应用的评测点主要包括:

- ❑ 通过存证记录接口, 提交存证数据至区块链系统, 在系统完成记账操作后, 可在数据层账本中检索到该存证记录;
- ❑ 通过存证检索接口, 将已存证的数据从区块链中检索, 可获得存证记录信息, 如存证记录所在块高度、时间戳等。

### 2. 溯源类应用

溯源类应用, 在信息记录的同时, 还需要将多条记录进行关联, 以便能够进行回溯。

具体实现上, 可以采用键值对的方式进行, 使用相同的 Key 进行关联; 也可以采用类似比特币 UTXO 模型进行关联和转移。

根据状态转移的方式还可以分为 4 类:

- ❑ 转移——从一条记录转到一套记录, 例如一对一转账;
- ❑ 分散——从一条记录转到多条记录, 例如一对多转账;
- ❑ 合并——从多条记录转到一条记录, 例如多对一转账;
- ❑ 混合——从多条记录转到多条记录, 例如多对多转账。

溯源类应用的评测点主要包括:

- ❑ 通过溯源登记接口, 提交溯源数据至区块链系统, 在系统完成记账操作后, 可在数据层账本中检索到该溯源记录, 包括当前数据信息、关联的上一条记录标识等;
- ❑ 通过溯源检索接口, 可以从区块链系统检索某一条溯源记录, 或检索某一数据的所有关联记录。

### 3. 资产类应用

资产 (Assets) 是指企业所拥有的, 用于从事生产经营活动以为投资者带来未来经济利益的经济资源。而数字资产 (Digital Assets) 则是资产的数字化形式, 通常与某种实际资产或虚拟资产相关联, 如数字代币等。

基于区块链的数字资产应用, 需要支持针对资产的全流程操作, 包括资产登记、资产发现、资产流转、资产撮合等。

资产类应用的评测点主要包括:

- ❑ 通过资产登记接口, 提交资产数据至区块链系统, 在系统完成记账操作后, 可在数据层账本中检索到资产记录;
- ❑ 通过资产转移接口, 提交资产转移请求至区块链系统, 在系统完成记账操作后, 可在数据层账本中检索到资产转移交易记录;
- ❑ 通过资产检索接口, 可以从区块链系统检索某一资产信息及其交易记录。

## 12.4 合约层评测

### 1. 功能评测

区块链系统如支持智能合约, 则需提供合约部署和合约能力, 其对应的评测点包括:

- ❑ 通过区块链系统接口申请部署合约, 部署成功返回合约地址, 通过该地址可访问合约定义的接口;
- ❑ 通过区块链系统接口部署已存在的合约, 提示合约已存在, 部署失败;
- ❑ 应用系统可调用合约接口, 合约按照预设逻辑执行并正确返回结果。
- ❑ 应用系统可对合约数据进行检索, 得到正确的返回结果;
- ❑ 合约执行事务性验证, 合约执行一半发生错误退出, 合约数据是否与执行前一致;
- ❑ 应用系统调用不存在的合约, 得到合约不存在的返回结果。

### 2. 性能评测

针对合约的执行、合约数据检索分别进行评测。

- ❑ 合约执行指对合约数据进行增加和修改操作的合约调用。
- ❑ 合约数据检索指对合约数据进行查询操作的合约调用。

其评测点包含以下几点。

- ❑ 对于合约线性执行的区块链系统, 验证单笔合约执行耗时, 可估算出合约执行的吞吐量。也可以通过工具进行压力测试, 得出合约执行吞吐量。
- ❑ 对于合约可并行执行的区块链系统, 在没有数据访问冲突的情况下, 通过工具进行压力测试, 得出合约执行吞吐量。
- ❑ 对于合约可并行执行的区块链系统, 在存在数据访问冲突的情况下, 通过工具进行



压力测试，得出合约执行吞吐量。

- 针对上述所有场景，在合约数据量增大的情况下进行测试，评估数据量的增长对合约执行吞吐量的制约关系。
- 使用性能测试工具调用合约精确数据检索接口，得出合约数据检索吞吐量；
- 使用性能测试工具调用合约模糊数据检索接口，得出合约数据检索吞吐量。
- 针对上述所有场景，在合约数据量增大的情况下进行测试，评估数据量的增长对合约数据检索吞吐量的制约关系。

### 3. 安全评测

合约的安全性测试，从合约调用、合约运行环境、数据隔离性等维度进行评测。

- 合约调用是否存在访问控制，如果有访问控制，使用无权限的账号调用合约，验证访问控制有效性。
- 合约运行环境，对于沙箱方式（如以太坊的 EVM），需单独对沙箱评估测试方案。如果区块链系统开源，还可采用专家判断和工具评测的方式对沙箱进行测试。
- 合约运行环境，对于合约在独立环境运行的方式（如超级账本 Fabric 0.6，每个合约单独部署在 Docker 中，通过 gRPC 调用），需要测试外部是否可以直接调用合约接口，如果调用不成功，则说明具备一定的安全性。
- 通过合约 A 的接口操作合约 B 中的数据，如果操作失败，则说明合约间的数据具备隔离性。

### 4. 可靠性评测

合约层可靠性测试主要针对合约异常调用场景进行验证。

- 合约执行过程中发生异常，不会影响其他请求的执行，不会影响区块链系统的正常运行。
- 可并行执行合约的区块链系统，通过性能测试工具对同一合约的同一数据进行并发访问，验证是否满足并发数据安全的特性。
- 合约执行过程中发生异常，合约的数据操作可保证事务特性。
- 调用不存在的合约，提示调用失败，合约不存在。

### 5. 兼容性评测

针对区块链系统版本升级或运行环境变更的兼容性。

- 对区块链系统版本进行升级，评测原合约是否可正常运行，原合约数据是否正确并可正确获取。
- 区块链系统部署环境变更，对于合约在独立环境运行的方式，验证合约是否可正确执行，如不行，评测哪些运行环境不兼容或需要修改哪些系统参数。

### 6. 可维护性评测

智能合约的可维护性，主要体现为：

- 是否可支持合约升级；
- 升级后的合约是否保留原合约的数据；
- 升级后的合约是否支持数据结构的变更，是否兼容原合约的数据结构；
- 是否可以将智能合约注销，以保证在智能合约发生严重逻辑缺陷时，可以终止该合约交易，避免造成更大的业务损失或数据错误。
- 智能合约执行日志是否可审计，合约的执行过程是否可通过日志跟踪。

## 12.5 激励层评测

激励层将经济学原理集成到区块链技术体系中来，它包括经济激励的发行机制和分配机制等。

- 在私有链和联盟链中，激励层不是必需的，因为参与记账的节点往往是在链外完成了博弈，通过强制或自愿的方式参与记账。
- 而在公有链中，必须引入激励层，来奖励遵守规则参与记账的节点，同时惩罚不遵守规则的节点，让整个系统朝着良性循环的方向发展。

激励层主要是基于博弈论（Game Theory）及纳什均衡（Nash equilibrium）模型来设计的。

博弈论又称为对策论，或者赛局理论，是应用数学的一个分支。1944年，冯·诺伊曼与奥斯卡·摩根斯特恩合著《博弈论与经济行为》，标志着现代系统博弈理论的初步形成。博弈论被认为是20世纪经济学最伟大的成果之一。博弈论考虑游戏中的个体的预测行为和实际行为，并研究它们的优化策略。表面上不同的相互作用可能表现出相似的激励结构，所以它们是同一个游戏的特例。

根据不同的基准，博弈有多种分类方式。一般来说，博弈可以分为合作博弈和非合作博弈；而从行为的时间序列性，可进一步分为静态博弈和动态博弈；按照参与人对其他参与人的了解程度，又可分为完全信息博弈和不完全信息博弈。

目前经济学家们所谈的博弈论主要指非合作博弈（Non-Cooperative Game），见表12-2。

表 12-2 非合作博弈分类及其对应的均衡模型

| 非合作博弈的分类  | 均衡模型                   | 英语术语                              |
|-----------|------------------------|-----------------------------------|
| 完全信息静态博弈  | 纳什均衡 <sup>①</sup>      | Nash equilibrium                  |
| 完全信息动态博弈  | 子博弈精炼纳什均衡 <sup>②</sup> | Subgame Perfect Nash equilibrium  |
| 不完全信息静态博弈 | 贝叶斯纳什均衡 <sup>③</sup>   | Bayesian Nash Equilibrium         |
| 不完全信息动态博弈 | 精炼贝叶斯纳什均衡 <sup>④</sup> | Perfect Bayesian Nash Equilibrium |

① 参见约翰·纳什的《非合作博弈》（1950）。

② [https://en.wikipedia.org/wiki/Nash\\_equilibrium](https://en.wikipedia.org/wiki/Nash_equilibrium)，纳什均衡。

③ 子游戏博弈精炼纳什均衡，[https://en.wikipedia.org/wiki/Subgame\\_perfect\\_equilibrium](https://en.wikipedia.org/wiki/Subgame_perfect_equilibrium)。

④ 贝叶斯纳什均衡，[https://en.wikipedia.org/wiki/Bayesian\\_game#Bayesian\\_Nash\\_equilibrium](https://en.wikipedia.org/wiki/Bayesian_game#Bayesian_Nash_equilibrium)。

⑤ 精炼贝叶斯纳什均衡，[https://en.wikipedia.org/wiki/Perfect\\_Bayesian\\_equilibrium](https://en.wikipedia.org/wiki/Perfect_Bayesian_equilibrium)。



如表 12-2 所示, 非合作博弈又分为 4 种: 完全信息静态博弈, 完全信息动态博弈, 不完全信息静态博弈, 不完全信息动态博弈。与之相对应的均衡概念模型分别为: 纳什均衡、子博弈精炼纳什均衡、贝叶斯纳什均衡、精炼贝叶斯纳什均衡。

以博弈论中经典“囚徒困境”为例: 以经济学中“理性经济人”的假设作为前提, 两个囚犯符合自己利益的选择是坦白招供。原本对双方都有利的策略, 双方都抵赖从而只被判刑 1 年的结果就不会出现。这样两人都选择坦白的策略以及因此被判 5 年的结局就是一个“纳什均衡”。表 12-3 所示为非合作博弈下“囚徒困境”的纳什均衡点。

表 12-3 非合作博弈下“囚徒困境”的纳什均衡点

| 博弈矩阵  | 囚犯甲招供          | 囚犯甲抵赖           |
|-------|----------------|-----------------|
| 囚犯乙招供 | 甲、乙均判刑 5 年     | 甲无罪释放, 乙判刑 10 年 |
| 囚犯乙抵赖 | 甲判刑 10 年、乙无罪释放 | 甲、乙均判刑 1 年      |

区块链的本质是一种多方参与并形成纳什均衡的共识系统。从工程控制的角度来讲, 区块链不是由管控方静态实现的, 而是通过设计一个“博弈场”来实现一种良性的博弈机制: 即遵守规则的参与者将获得利益, 而破坏规则必将会受到制裁出局。

纳什的非合作博弈论模型仅仅是突破了博弈论中的一个局限。一个更大的局限是, 博弈论面对的往往是由几十亿节点的庞大对象构成的社会、经济等复杂行为, 但冯·诺伊曼和纳什的研究是针对两三个节点的小规模博弈论。近期的研究成果, MIT 的一位计算机科学博士生的博士论文认为“找到纳什均衡点是几乎不可能的事”<sup>①</sup>。

因此, 理论分析很难有效或全面地分析出激励层的优劣。最好的评测方法是在实际运行场景中进行验证, 通过参与系统各方的实际行为分析得出。例如: 在比特币网络中, 任何节点拒绝合作都是无利可图的, 而只要大多数节点选择合作, 那么任何节点选择合作都是唯一理性的选择。

专家评测可以作为激励层评测的补充方法。在区块链系统中, 可以按照记账节点、非记账节点、客户端节点、合约节点等角色划分, 将庞大的社会经济问题简化为几个角色间的博弈问题, 并进行理论分析, 判断其激励机制的有效性。

考虑到大多数私有链和联盟链都不具备激励层, 因此, 在本章中不针对激励层单独设计评测点。

## 12.6 共识层评测

共识问题 (Consensus Problem) 是分布式系统必须解决的一个问题。一个严格的共识算法需要满足如下几个条件。

1) 终止性 (Termination): 所有的执行过程最终会做出一个决定。它要求算法必须在有

<sup>①</sup> The Complexity of Nash Equilibria. <http://people.csail.mit.edu/costis/thesis.pdf>, 2008.

限时间内结束，不能无限循环下去。

2) 一致性 (Agreement): 每个正确的执行过程应该在相同的值上达成一致，表示我们期望获得相同的决议。

3) 合法性 (Validity): 如果所有正确的执行过程提出了相同的值 V，那么所有正确的执行过程都会决定值 V，这可排除进程初始值对自身的干扰。

4) 诚实性 (Integrity): 每个正确的执行过程最多只能决定一个值，如果它决定了某个值的话，这个值一定是被某个执行过程提出的。

□ 如果全部执行过程都是正确的，那么该算法为“非拜占庭容错”共识算法。

□ 如果存在少数不正确的执行过程，那么该算法为“拜占庭容错”共识算法。

基于以上要求，可以推导出在分布式系统领域非常重要的定理“FLP 不可能性” (FLP impossibility): 在假设网络可靠、计算节点只会因崩溃而失效的最小化异步模型系统中，仍然不存在一个可以解决一致性问题的确定性算法。<sup>①</sup>

两将军问题 (Two Generals' Problem) 也表明，通过不可靠网络通信的两个进程不可能达成一致的决策。<sup>②③</sup>

因此，现有的共识算法，通常都是假设在一个同步通信场景中进行的。表 12-4 为共识算法的分类。

表 12-4 共识算法的分类

| 序号 | 诚实性 | 拜占庭容错 | 终止性 / 确定性 | 常见共识算法                 | 典型场景  |
|----|-----|-------|-----------|------------------------|-------|
| 1  | 是   | 否     | 确定性一致     | PAXOS、RAFT 等           | 企业私有链 |
| 2  | 否   | 是     | 确定性一致     | PBFT、Q/U、Zyzyva、RBFT 等 | 行业联盟链 |
| 3  | 否   | 是     | 大概率一致     | PoW、PoS、DPoS 等         | 公有区块链 |

如表 12-4 所示，在实际应用中，不同类型的区块链会采用不同类型的共识算法，需要区别对待。

1. 功能测试

不同类型的共识算法，根据其实现方式不同，评测要求会有差别。对于共识算法的评测，首先应当进行专家判断，通过算法分析判断其是否支持拜占庭容错、故障容忍率、出块速度、块确认时间等指标的理论值。在专家判断的基础上，通过手工测试和脚本测试的方法，对共识算法的逻辑进行验证。

除了正常情况的验证外，还有异常情况时的分叉、同步等问题，也需要单独设计评测点。

1) 分叉：对于确定性共识算法，共识过程中每一块都是立即确认，因此不会出现分叉

① Fischer, Lynch, Patterson. Impossibility of Distributed Consensus with One Faulty Process, 1985.  
② Akkoyunlu, Ekanadham, Huber.《网络通信设计中的一些约束和权衡》(73 页), 1975:73.  
③ Jim Gray. 数据库操作系统笔记 . 1978:465.



的问题。而对于一部分大概率共识算法，如 PoW，由于其共识算法的特殊性，各记账节点同时竞争出块，很可能导致链分叉的情况。对于哪条链是有效的链，通常采用多块确认的方式。

2) 同步：节点的块高度落后于其他节点时，该节点需要从其他节点同步块数据。块高度落后可能由 2 种原因导致：共识过程处理缓慢，导致进度滞后；节点因为某种因素（故障、升级等）导致块高度落后。

总的来说，本部分的主要评测点包括：

- ❑ 在无故障、无欺诈节点时，能在规定时间内达成一致的、正确的共识，输出正确结果；
- ❑ 故障容忍率范围内，能在规定时间内达成一致的、正确的共识，输出正确结果；
- ❑ 在故障容忍率范围外，无法在规定时间内达成一致的、正确的共识，输出正确结果；
- ❑ 对于交易重放的情况，能够识别该交易并执行失败；
- ❑ 存在欺诈节点的情况，如果存在被篡改的交易，能够识别该交易并执行失败；
- ❑ 在共识产生分叉时，所有节点可以对哪条链是主链达成一致的、正确的共识；
- ❑ 节点在发现块高度落后于其他节点时，可自动发起块同步操作，以保证尽快恢复到与其他节点一致的状态；
- ❑ 同步过程中，可对接收的块进行有效性验证。

## 2. 性能测试

共识性能一般通过平均出块时间、交易吞吐量、区块确认时间等来评估。

- ❑ 平均出块时间。使用工具持续对区块链系统进行测试，并获得一定时间内产生的区块总数来计算平均出块时间。
- ❑ 交易性能（TPS）。交易通常指需要写入区块中的事务性操作，因此，可通过性能测试工具对系统进行压力测试，通过出块总时间和总交易笔数估算平均交易吞吐量。根据不同区块链方案和节点数的组合，进行多次测试，将收集到的数据汇总到同一个数据文件中，据此，可以得到类似表 12-5 所示表格形式的数据。多次的测试结果，还可以进一步计算并获得其均值、中间值、最大值和最小值等。

表 12-5 区块链交易吞吐量的测试结果示例

| 共识节点数   | 第一次测试 | 第二次测试 | 第二次测试 | 均值 |
|---------|-------|-------|-------|----|
| 7 节点    |       |       |       |    |
| 19 节点   |       |       |       |    |
| 31 节点   |       |       |       |    |
| 121 节点  |       |       |       |    |
| 1201 节点 |       |       |       |    |
| ...     |       |       |       |    |

注意：针对无须写入区块的查询操作性能（QPS），和传统集中式软件的性能测试手段一致。由于这类交易可以在区块链之外通过缓存等手段来提高其性能，与区块链系统本身

无直接关系，因而不单独设计评测点。

- ❑ 针对确定性一致性的共识算法，如 PAXOS、RAFT、PBFT、Q/U、Zyzyva、RBFT 等，由于出块是立即确认，其区块确认时间等同于出块速度。
- ❑ 针对大概率一致性的共识算法，如 PoW、PoS、DPoS 等，由于存在分叉的可能，其区块确认时间需要根据共识算法的原理，由专家判断方式得出其理论值。

### 3. 安全性测试

共识层的安全性方面，主要针对拜占庭故障的容错能力进行评测。

首先通过专家判断确定系统所提供的公式算法是否支持拜占庭故障。如果支持拜占庭故障容错，则可参照下述方法在故障容忍度之内进行评测。如果交易可正确处理，则说明支持拜占庭容错能力。

- ❑ 任意节点错误响应，包括执行成功对外返回“失败”，执行失败或者不执行的时候对外返回“成功”。
- ❑ 任意节点错误响应，包括广播的消息与接收的交易不一致。
- ❑ 任意节点打包时篡改交易记录，如交易数据、签名数据等。

### 4. 可靠性测试

主要通过模拟节点宕机故障、网络故障来评测区块链系统是否可以容错并正常工作。

- ❑ 在共识算法故障容忍度范围内，将节点关闭，持续进行交易，经过一段时间后启动，整个过程中区块链系统可正常工作，故障节点启动后可恢复并参与交易。
- ❑ 将超过共识算法故障容忍度节点数的节点关闭，区块链系统无法完成共识；再将故障节点启动，区块链系统可恢复交易。
- ❑ 在共识算法故障容忍度范围内断开网络连接，验证区块链系统是否可正常工作。
- ❑ 在共识算法故障容忍度范围内随机模拟网络链路丢包，验证区块链系统是否可正常工作。
- ❑ 在共识算法故障容忍度范围内随机模拟网络延时（抖动），验证区块链系统是否可正常工作。
- ❑ 模拟超过共识算法故障容忍度范围节点的网络故障（如脑裂问题），区块链系统无法完成共识；再将故障节点的网络恢复，区块链系统可恢复交易。

### 5. 可维护性测试

主要从共识参数设置、节点可扩展、共识状态可监控 3 个方面来设计评测点。

- ❑ 按照系统的操作手册（离线或在线），对系统共识算法的参数进行设置，并评测验证是否生效。例如（最长 / 平均）出块时间、块大小（单位为交易笔数或物理大小）、共识算法类型、节点数、共识节点列表等参数。
- ❑ 按照系统的操作手册（离线或在线），对系统共识算法进行切换，评测区块链系统的共识算法是否可切换，是否可正常工作。



- ❑ 首先通过专家判断,分析系统所支持的共识算法是否可支持节点动态扩展。
- ❑ 通过手工测试或脚本测试,在无故障的环境下,按照系统的操作手册正确的增加或删除节点,区块链系统可正常工作。
- ❑ 可通过系统接口(服务接口或命令行接口)实时获取共识节点的运行状态。例如,块高度、共识算法类型、是否为主节点(涉及确定性共识算法)、难度系数(涉及 PoW 共识算法)等。

## 12.7 网络层评测

### 1. 功能测试

- ❑ 设定节点的类别和能力,包括设定该节点是记账节点(验证节点、共识节点)还是非记账节点(非验证节点),以及该节点所拥有的权限。
- ❑ 记账节点,能否动态地加入或退出区块链网络,而其他节点不受影响。

### 2. 性能测试

- ❑ 查看从节点发出请求到收到响应的平均延时,如在高并发情况下,从客户端发起请求,运行 30 分钟后计算平均请求延时时间,并记录节点数和平均响应时间。
- ❑ 增加节点对网络延时的影响,如在上面的基础逐渐增加节点,看网络平均响应延时,并记录增加节点数和平均响应时间。
- ❑ 增加节点对吞吐量的影响,如在第一条基础上增加节点,查看吞吐量变化。

### 3. 安全性测试

- ❑ 查看增加或者删除节点时是否有证书密钥验证,如增加节点和删除节点时,日志里面是否有证书密钥验证,防止任意伪造节点接入。
- ❑ 查看区块链网络抗攻击能力,如抗 DDOS、女巫攻击、黑客攻击的能力。

### 4. 可靠性测试

- ❑ 当前记账节点失效时,能否自动切换到其他节点进行记账,交易请求不会丢失。
- ❑ 模拟网络限速、网络断开、包丢失、包重复、包抖动等故障,检查此时区块链网络和服务的健康状态,是否还能正常服务。

## 12.8 数据层评测

### 1. 功能测试

- ❑ 检查区块链共享账本的数据结构是否和其所描述的相符。如:比特币账本中的交易结构基于 UTXO 模型。
- ❑ 验证区块大小能否调整最大值,是静态调整,还是动态调整。前者需要重启全部网

络节点，后者则无须此步骤。观察该调整后，参数是否生效，且不影响整个区块链网络的数据完整性。

❑ 是否支持账本下载功能，包括全量账本下载、部分账本选择性下载。

## 2. 性能测试

在评测共享账本性能时需要考虑拓扑结构、节点总数、网络带宽等因素。

❑ 统计多节点账本同步延时，如：统计区块链网络节点的账本同步延时是否达标，是否受其他方面影响，要是有影响，并定位瓶颈。例如节点暂停 10 分钟，看 10 分钟内节点快是否能同步上，暂停 20 分钟，记录多长时间内能够同步上。

❑ 账本数据总量是否对同步延时有影响，确定最佳账本大小。如分别针对账本数据大小为 1GB、10GB、20GB 时，记录同步延时长，并定位瓶颈。

❑ 修改账本中单个区块大小是否对同步延时有影响，确定最佳账本区块大小。例如：区块大小为 1MB 或者 1000 条数据，在账本 100GB 情况下，记录同步延时时间，并定位瓶颈。

## 3. 安全测试

❑ 查看共享账本数据访问控制权限。例如，在私有链和联盟链里面根据账户权限验证账本数据访问权限，访问数据是否水平越权和垂直越权。

❑ 查看共享账本交易数据是否加密。例如，在私有链和联盟链里面共享账本交易数据有没有加密。

❑ 查看共享账本交易数据是否支持选择性加密。例如，在私有链和联盟链里面共享账本交易数据支持选择性加密。

❑ 查看共享账本哪些数据用到签名。例如查询交易、转账交易是否签名。

❑ 查看交易有没有支持匿名。

❑ 查看共享账本数据是不是不可篡改。

## 4. 可靠性

❑ 查看共享账本服务稳定性。共享账本是区块链的核心，只有保证区块账本充分成熟稳定，才能保证共享账本提供 7×24 小时服务。

❑ 查看单节点故障导致服务失效，恢复时是否能容错同步区块。例如，一个节点服务异常操作（这里的异常指杀掉进程、掉电、删除一些块、发送异常包数据以及硬件 IO 故障等），启动服务，看账本区块是否能容错同步。

❑ 查看单节点程序中断或失效时，恢复时区块账本是否可用。例如，当一个节点服务账本中的 IO 超时、IO 突然降级，恢复时区块账本是否可用。

## 5. 可移植性测试

账本数据结构是否支持移植到别的链中使用，例如，当另外的链需要此数据时，此链的账本数据结构可移植到另外的链的节点上，是否能使用。



## 6. 可维护性测试

- ❑ 账本数据结构升级，账本数据是否可用。例如，当系统变更、升级前后，看本节点账本数据服务是否提供服务。
- ❑ 账本数据结构升级是否影响到另外一个节点账本使用，如系统变更、升级前后，是否影响到另外一个节点账本数据服务提供服务。

## 12.9 辅助工具

工欲善其事必先利其器，选用一些好的测试工具，往往有意想不到的效果。下面就介绍几种用于区块链测试的网络模拟、错误注入、模糊测试等工具，以及一个可参考的区块链评测框架。

### 1. 网络模拟工具

传统软件产品测试通常是在性能良好的局域网中进行，只考虑了连接超时（Connect Timeout）、读超时（Read Timeout）、连接异常断开（Broken Pipe）等一些较为简单的网络异常。而对于类似区块链这种分布式系统而言，这些网络异常是远远不够的，还需要模拟出广域网下各种复杂的网络异常。

此时，我们需要借助一些网络模拟工具（Traffic Control）来设置网络带宽（bandwidth）、网络延迟（latency）、丢包率（packet loss）、错包率（corrupted packets）、乱序率（packets ordering）等参数，可以模拟低带宽、高延迟、包丢失、包重复、包损坏、包抖动、包乱序等网络异常。

#### 网络模拟工具 1——TC（Traffic Control）

TC 是 Linux 内核自带的网络流量控制器，它利用队列规定建立处理数据包的队列，并定义队列中的数据包被发送的方式，从而实现对流量的控制。

典型场景的 TC 指令示例如下：

- ❑ 模拟限速：/sbin/tc qdisc add dev eth0 root tbf rate 2048kbit latency 50ms burst 2048
- ❑ 模拟包丢失：/sbin/tc qdisc add dev eth0 root netem loss 10%
- ❑ 模拟包重复：/sbin/tc qdisc add dev eth0 root netem duplicate 1%
- ❑ 模拟包损坏：/sbin/tc qdisc add dev eth0 root netem corrupt 0.2%
- ❑ 模拟包乱序：/sbin/tc qdisc add dev eth0 root netem delay 1000ms 300ms 30%

#### 网络模拟工具 2——ATC（Augmented Traffic Control）<sup>①</sup>

ATC 是由 Facebook 公司开源的移动网络测试工具，它有 2 个最吸引人的特点：

- ❑ 在手机上通过 Web 界面就可以随时切换各种不同的网络环境；
- ❑ 多个手机可以连接到同一个 WiFi 下，相互之间模拟的网络环境各不影响。

<sup>①</sup> <http://facebook.github.io/augmented-traffic-control/>, A tool to simulate network conditions.

## 2. 故障注入工具

相对于传统软件，由于分布式软件系统涉及的节点更多、环境更复杂，所以某些故障很难发生。软件修复此故障后，由于该故障很难重现，也无法验证是否修复成功。所以，我们需要采用“故障注入”(Fault Injection)的方式强制该故障出现。

故障注入法技术有多种实现方式。

❑ 基于硬件的故障注入。

❑ 基于软件的故障注入：通过修改底层驱动软件、操作系统的动态库、特定软件接口等，来模拟各种系统故障，如 CPU 中断、内存故障、IO 故障、网络故障等。

❑ 基于模拟的故障注入：这种技术使用在模拟测试的计算机原型上，通过在模拟过程中，改变逻辑值来达到故障注入的效果。

❑ 混合故障注入技术：综合采用多种故障注入技术，取长补短。

❑ 有多种工具可以帮助实现错误注入。

下面介绍几个开源的故障注入工具。

### 故障注入工具 1——DICE-Fault-Injection-GUI<sup>①</sup>

它是一款开源的错误注入工具，可通过 SSH 远程命令方式，人为地在虚拟机层植入一些硬件故障。在此基础上，它封装了一个 Web 应用。通过将其源码用 JDK1.8 编译成 WAR 并部署在 Tomcat 上，就可通过浏览器界面进行相关操作，来模拟 CPU 负荷过大、内存不足、磁盘 I/O 繁忙等故障。其界面如图 12-8 所示。

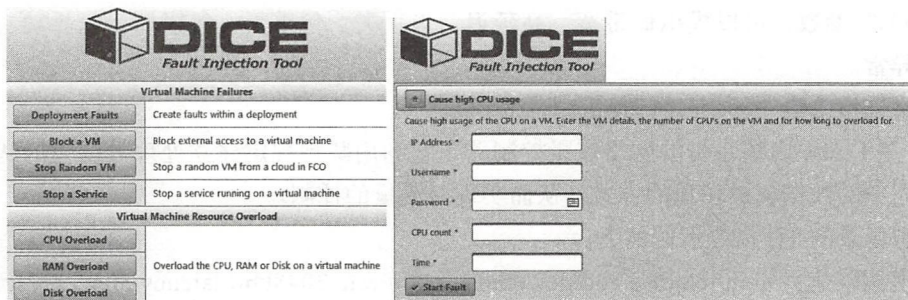


图 12-8 故障注入工具 DICE-Fault-Injection-GUI

### 故障注入工具 2——Jepsen<sup>②</sup>

Jepsen 是由 Kyle Kingsbury 采用函数式编程语言 Closure<sup>③</sup>编写的分布式系统的一致性验证框架。作者使用它针对许多著名的分布式系统，如 MySQL-cluster、ZooKeeper、MongoDB、ElasticSearch 以及 Redis 等，进行了一致性验证，并且成功发现了很多分布式系统的深层缺陷。

① <https://github.com/dice-project/DICE-Fault-Injection-GUI>, DICE-Fault-Injection-GUI.

② <https://github.com/jepsen-io/jepsen>, A framework for distributed systems verification, with fault injection.

③ <https://clojure.org/>, The Clojure Programming Language.



在 Jepsen 的源码中,可以找到针对 MySQL-cluster、ZooKeeper、MongoDB、ElasticSearch 以及 Redis 等系统的测试代码,可以在此基础上进行扩展,来针对特定的分布式系统进行测试验证。

### 故障注入工具 3——Namazu<sup>①</sup>

Namazu 是一个用于分布式系统测试的可编程的模糊测试调度框架,曾成功找出 ZooKeeper、Etcd、YARN 的几个缺陷,包括 ZooKeeper-2212、ZooKeeper-2080、etcdctl bug #3517、YARN-4301 等。

## 3. 模糊测试工具

完全的手工测试即“渗透测试”,由测试人员模拟黑客恶意进入系统、查找漏洞。所有渗透测试对人员能力的依赖性强、成本高,难以大规模的实施。而模糊测试(Fuzz Testing)<sup>②</sup>是一种介于自动或半自动的软件测试技术。它通过向计算机程序的输入提供无效、意外或随机数据,然后监视程序是否有异常,例如崩溃、内置代码断言失败或查找潜在的内存泄漏。

模糊测试的技巧在于它是不符合逻辑的:自动模糊测试无须像测试人员那样,去猜测哪个数据会导致系统故障,而是将尽可能多的杂乱数据输入到程序中,模糊地触发并帮助快速彻底地发现错误。这种技术可以用来确保流行的开源组件和关键的 IT 基础设施是稳定、安全、可靠的。

这项技术,对于区块链这类基础设施类软件,尤为重要。

### 模糊测试工具 1——Go-Fuzz<sup>③</sup>

Go-fuzz 是一个采用 Golang 语言编写的、开源的模糊测试工具。通过半自动或自动地为程序提供非法的、非预期、随机的数据,并监控程序在这些输入数据下的故障、内置断言、内存泄露等情况。它生成用例采用是遗传算法,非常符合数据生成方式。

### 模糊测试工具 2——OSS-Fuzz<sup>④</sup>

OSS-Fuzz 是由 Google 公司开源的测试版安全工具,能够针对开源软件进行持续的模糊测试。它的目的是利用更新的模糊测试技术与可拓展的分布式执行相结合,提高一般软件基础架构的安全性及稳定性。OSS-Fuzz 结合了多种模糊测试技术/漏洞捕捉技术(原 libfuzzer)与清洗技术(原 AddressSanitizer),并且通过 ClusterFuzz 为大规模可分布式执行提供了测试环境。

## 4. 其他参考工具

### 工具——私有区块链的分析框架 BlockBench<sup>⑤</sup>

BlockBench 是由新加坡国立大学和浙江大学开发的私有链评估框架,它针对目前最成

① <http://osrg.github.io/namazu/>, A programmable fuzzy scheduler for testing distributed systems.

② <https://en.wikipedia.org/wiki/Fuzzing> 模糊测试.

③ <https://github.com/dvyukov/go-fuzz/>, Go-Fuzz: Randomized testing for Go.

④ <https://github.com/google/oss-fuzz>, OSS-Fuzz - continuous fuzzing of open source software.

⑤ <http://www.comp.nus.edu.sg/~dbsystem/blockbench/>, BLOCKBENCH: A Framework for Analyzing Private Blockchains.

熟的、能够支持智能合约功能的3个区块链平台（Hyperledger Fabric、Ethereum、Parity）进行设计，以实现对其评估。该框架还将准备广泛地支持未来的区块链平台。

它可以度量5个关键的标准。

- 吞吐量：测量每秒钟成功交易的数量。
- 延时：测量每个交易的响应时间。
- 可扩展性：测量增加节点和并发工作负载时吞吐量和延时的变化。
- 容错能力：测量节点故障期间和响应时间如何变化，例如停止故障、网络延时和任意消息的错误情况。
- 安全性：模拟网络分区攻击，测量区块堵塞率。

## 12.10 小结

本章从区块链系统整体出发，采用正交分析法，从6个层面和8大类质量指标来设计评测点和测试用例，如表12-6所示。

表 12-6 区块链测试用例的分层和分类

| 层次 | 功能      | 性能      | 安全性     | 可靠性     | 兼容性     | 易用性 | 可移植 | 可维护     |
|----|---------|---------|---------|---------|---------|-----|-----|---------|
| 应用 | 1 ~ 2   |         |         |         |         |     |     |         |
|    | 3 ~ 4   |         |         |         |         |     |     |         |
|    | 5 ~ 7   |         |         |         |         |     |     |         |
| 合约 | 8 ~ 13  | 14 ~ 20 | 21 ~ 24 | 25 ~ 28 | 29 ~ 30 |     |     | 31 ~ 35 |
| 激励 |         |         |         |         |         |     |     |         |
| 共识 | 36 ~ 43 | 44 ~ 47 | 48 ~ 50 | 51 ~ 56 |         |     |     | 57 ~ 61 |
| 网络 | 62、63   | 64 ~ 66 | 67、68   | 69、70   |         |     |     |         |
| 数据 | 71 ~ 73 | 74 ~ 76 | 77 ~ 82 | 83 ~ 85 |         |     | 86  | 87 ~ 88 |

从表12-3中也可看出，本章的评测点尚未覆盖所有的栏目。另一方面，部分栏目不容易进行统一评测，例如激励层更多的是经济学原理而非技术性实现。还有，测试用例本身也很难穷尽，需要在权衡测试成本和测试效率等因素。本章只列出笔者认为相对重要而且可测试性较佳的一些评测点，但也不可避免地会有所疏忽或遗漏。读者可参考上述表格所示的正交分类，进一步补充相关评测点和测试用例。

未来的区块链评测的发展方面，在测试内容上，短期还会出现针对不同行业的差异性，但最终会走向融合和统一。而在测试手段上也会多样化，通过结合故障注入、模糊测试、基准框架等工具，将逐步由人工测试转向自动化测试。



# 后 记

作者：邹均

对于技术书籍而言，特别是像区块链这种比较新的领域来说，很少有人能从头到尾看完书里的所有章节。如果连后记都看，那就是真正碰到对区块链有感情的读者，碰到知音了。在此，也对读者的学习精神表示深深的敬意，并对读者由始至终的支持表示感谢。最后，也希望借后记，和读者再交流一下关于区块链过去、现在和未来发展的话题。

区块链有两个比较有特色的点。一个是链上数据不可篡改、链上的交易信息可追溯。另一个点是对等节点在陌生环境通过共识机制建立信任，建立协同。我们发现一个很有趣的地方：区块链这两个特点和我们人类都有关联。首先，我们知道，人类男性细胞核里的Y染色体是单性遗传，沿着Y染色体的遗传链，我们可以追溯到6万年前人类共同的父亲；而人类女性细胞质里的线粒体上的37个基因，也是单性遗传，沿着线粒体的37个基因遗传链，我们可以追溯到10万年前人类共同的母亲——线粒体“夏娃”。这个人类遗传链，和区块链何其相似。与通常区块链上记录交易信息或其他状态信息不同，在人类区块链上记录的是基因的遗传信息。

而另一方面，就像共识在区块链上起决定性作用一样，共识也在人类的演化和发展上起着决定性的作用。著名以色列作家尤瓦尔在《人类简史》中提出一个非常新颖的观点。他认为人类的祖先智人在大约10万年前，发生了一场认知革命。这场革命始于智人自发产生的一种抽象思维能力，能虚构出很多概念、故事，例如像“神”“爱”“宗教”等。这些智人在这些虚构的概念和故事中形成共识，最后能在共识下形成群体协作，结果就是打败了当时比智人更强大的直立人、猛兽，一举从食物链的中端跃居到食物链的顶端。

区块链的核心是通过共识机制在一个分布式、对等的网络中的节点确定网络状态。因此，它可以作为未来大规模生产协作的基础设施。而未来的生产协作，将是人与人之间、人与智能设备之间、智能设备与智能设备之间的大规模生产协作。

这种协作，在未来的数据时代显得更为重要。一个重要的原因，是因为在数据时代，人必须依赖智能设备才能对大规模数据进行分析、处理。

尤瓦尔在完成《人类简史》之后，又出版了《未来简史》。书中也不乏惊世骇俗的论断，其中的一段说的是人类的主流意识形态正从人文主义向数据主义转换。他接着分析，“数据

主义认为，宇宙由数据流组成，任何现象或实体的价值就在于对数据处理的贡献。”这个看似荒唐的结论却能从现实社会找到佐证。例如，马云认为现在已经不是 IT 时代，而是 DT 时代；阿里巴巴是大数据公司而不是零售公司。而另一方面，用数据说话，相信数据而不是相信人的一面之词已经成为当今商业社会运转的准则。互联网平台公司已经从争夺流量变成争夺数据资源。十九大报告也专门提到“数字经济”、“数字中国”。数字经济成为继农业经济、工业经济之后新的经济形态，正深刻改变着人类生产生活方式。

数字经济是以数据作为关键生产要素，以互联网作为重要载体，以云计算、大数据等 IT 技术为核心驱动力，通过数字经济在农业、工业、服务业、公共服务等重要领域深度融合应用，重塑经济社会发展与治理模式的新型经济形态。数字经济时代和以往的农业经济、工业经济有很大的不同。首先，在农业经济和工业经济时代，生产力是获取物质资料的能力，生产关系中关注的是物质的所有权和使用权；而在数字经济时代，生产力是获取数据、处理数据的能力，生产关系中关注的是数据的所有权和使用权。

数字经济时代和我们熟悉的信息革命时代也不一样。信息革命时代是自计算机出现后人类将信息处理融于经济活动，其特点是人类将数据转化成信息，信息转化成知识，最后知识转化成智能。而到了现在，积累的数据量从量变到质变，已经大到超出了人处理的能力，只能依赖机器智能来处理日益增长的数据，从而引发了数据革命，进入数字经济时代。

在数字经济时代，数据是关键的生产要素。如何为数据确权，保障数据权益呢？如何实现数字资产安全在线上的转移？在信息革命时代，甚至互联网时代，没有很好的解决办法。互联网时代趋于中心化的业务模式也使得数据垄断，数据权益被侵犯。而在进入数字经济时代的时间节点，区块链技术应运而生，为解决数据确权、数字资产价值安全线上流转、数据权益保护提供了良好的技术手段。

很多人都知道区块链是信任机器，能够在陌生环境中建立信任。但信任什么？如何建立信任，可能会有不同理解。我们认为，区块链上建立的信任，是对基于共识协议建立的对网络状态的信任。而线上数字资产转移，最重要的是解决“双花”问题。传统信任是任何经济活动的基础。经济活动是围绕着契约进行，信任是契约能否履行的一个主观判断。在数字经济时代，最重要的是保证数字资产能安全转移，不存在“双花”的问题。数字资产需要用一個区块链上的 token 来唯一指代，信任将凝聚在 token 上。随着区块链与实体经济的结合，新的经济形式会出现，国外甚至有一个新的单词叫 tokenomics，在国内被称为“通证经济”。通证经济指的是采用 token 来做经济体中价值交换以及产品服务消费的媒介经济体。在通证经济中，token 可以有多种角色，例如可以是权益类 token，包括股权类、债权类、资产类、使用权 token，也有些带有虚拟货币属性，例如比特币。token 的各种角色在不同国家会有不同的合规性问题，存在一定的法律合规性风险。但通证经济在数字经济时代将起重要作用，原因是通证经济能解决数据确权、数据价值安全流转和数据价值变现等问题，同时也能使企业生态更好地形成闭环，拉近客户、投资者和供应链上合作伙伴的关系，提升企业生态上的价值流转效率。



除了把区块链上升到经济学层面，我们也看到很多学者把区块链提升到哲学层面。我们认为，区块链是我们看到的第一个能链接世界三元组的一个工具。在柏拉图看来，世界是由物理世界、精神世界和形式世界的世界三元组构成。近代哲学家波普尔则认为形式世界不好理解，把它改成了知识和信息世界。现代比较流行的观点是，世界三元组由物理世界、精神世界和结构世界组成。区块链本身来源于结构世界，它的共识机制、链式结构就是一个很好的佐证。但区块链提供的是保证精神世界里的价值守恒的机制，而它又需要有物理世界的支撑，包括运行区块链的物理服务器，共识所需的挖矿能耗等，都离不开物理世界。因此，区块链的意义在于，它可以链接这三个世界，同时给价值守恒提供保障，使得信息互联网可以向价值互联网过渡。

就区块链而言，它可能对回答哲学上的终极问题（“我是谁，我从哪里来，我到哪里去”）没有帮助，但对什么是真实客观性的问题会提供一个新的角度的答案。我们平时所说的“眼见为实”是一个过于简单，经不起推敲的说法。什么是真实，有真正意义上的客观世界存在吗？在这些问题背后隐藏的不单是哲学问题，也是深刻的物理学问题。现代物理学的发展，特别是在量子力学方面的发展，已经颠覆传统经典物理学中对时空的很多认知。著名物理学家霍金在其著作《大设计》中提出一个论点，认为不存在与图像或理论无关的实在概念。这里的图像或理论实际上是一个具有数学性质的模型，以及一组将这个模型的元素与观测相连接的规则。在霍金看来，世界是依赖于模型的存在。他举的例子是，在鱼缸里的金鱼建立的世界模型和鱼缸外的人建立的模型会很不一样。我们看到的直线运动对鱼缸里的金鱼来说就是曲线运动。不能简单地说是我们的模型真实过金鱼的模型。因为我们也无法确定是否有一个我们自己察觉不到的鱼缸存在，把我们所看到的图像扭曲；就像金鱼不知道它所看到的图像被鱼缸扭曲一样。说到底，人们看到的形象也是视觉信号在人脑中构建的一个三维模型。我们不能确定在大脑中构建的三维模型就是与实际客观存在相符。而客观存在也只能通过观察被认识，因此我们无法将客观存在和观察者绝对分开。

在霍金看来，判别一个模型的好坏，取决于以下几点：

- 1) 它是优雅的；
- 2) 它包含很少任意或者可调整的元素；
- 3) 它和全部已有的观察一致并能解释之；
- 4) 它可对将来的预测做详细的预言。如果预言不成立，观测就能证伪这个模型。

区块链也可以看成是一个模型。首先这个模型是优雅的，因为它具备简单性、不可篡改性和可追溯性。区块链上可调整的元素也很少，部署之后基本固定。它对将来的预测主要在于资产的安全转移和不可双花。在这个模型中，可以简单地提炼成一句类似笛卡儿所说的“我思故我在”——“共识即存在”。经过共识确认的信息，在区块链的模型上是可以被看成是真实的。至于它是不是真正客观真实的，可能不重要。重要的是区块链系统的节点根据共识协议，都认定共识后的结果是真实的。我们过往谈的真实，往往带有中心化和主观的意思在里面。如果从哲学层面上深究，其实不可能找到所谓的真实。区块链给我们

一个简单的模型，在这个模型中，真实就是经共识规则确认的状态。

区块链上依赖于模型的真实性的要延伸到链下也会遇到很大的挑战。传统的所谓真实性依赖于中心化的权威机构。例如当法官判案的时候，双方辩护人提供各自的证据链，最后由法官来根据双方提供的证据链，还原一个所谓的以证据拼接的真实，并依此判案。所以在现实世界也没有绝对的真实。同样，在区块链上也没有绝对的真实，区块链上共识的结果可以看成是真实的，就像法官判案，只不过区块链是以去中心化的方式来确认真实。分布式预言机（Oracle）是将链上依赖于模型的真实性的延伸到链下依赖于模型真实性的一个好办法。

区块链的分布式共识能力意味着它能提供一种规则性的技术，类似一个判案的法官，能起一个客观、公正的仲裁作用。这对人类社会来说意义重大。自有人类文明以来，不同种族、不同的文化、不同信仰的人们都有一个共同的期望，就是都希望社会是公平、公正的，人类生而平等，人们能按其自由意志来生活。但现实往往不是这样。笔者曾经在一个场合，听到一个朋友发自肺腑、发人深省的大愿，就是希望今后人们不再互害，不再出现养鱼的不吃自家养鱼，养鸡的不吃自家养的鸡的现象。他希望有人能帮助他实现这个愿望。区块链本身是一个很好的帮助他实现这个愿望的技术，因为它能够带来共识、带来规则、带来公正和公平。但我们也看到，目前区块链虽然很火热，但借区块链之名来“割韭菜”、诈骗、传销等现象不在少数。这样的现象任其发展很容易毁掉区块链。因此，作为一个区块链从业者，也借本书的后记发愿，期望尽个人绵薄之力，联合志同道合的同行，建设一个健康的区块链行业自律环境，用区块链的技术构建一个诚信的商业体系和社会。

最后，大家都会关心区块链未来将如何发展。根据霍金的观点，我们要回答这个问题，必须首先建立一个模型，然后通过观测的结果来和根据该模型做的预测来对比。如果观测与预测总是一致，那么我们就能够比较准确地预见未来区块链的发展，否则也只能是主观猜测。而要建立这样一个模型，不单需要对区块链技术的透彻理解，也需要掌握社会学、经济学方面的知识体系。现在由于区块链技术还处于起步阶段，建立这个模型还不现实。但期望有一天这个模型能被建立而且期望这个模型的建立者会是我们的读者之一。



## 内容简介

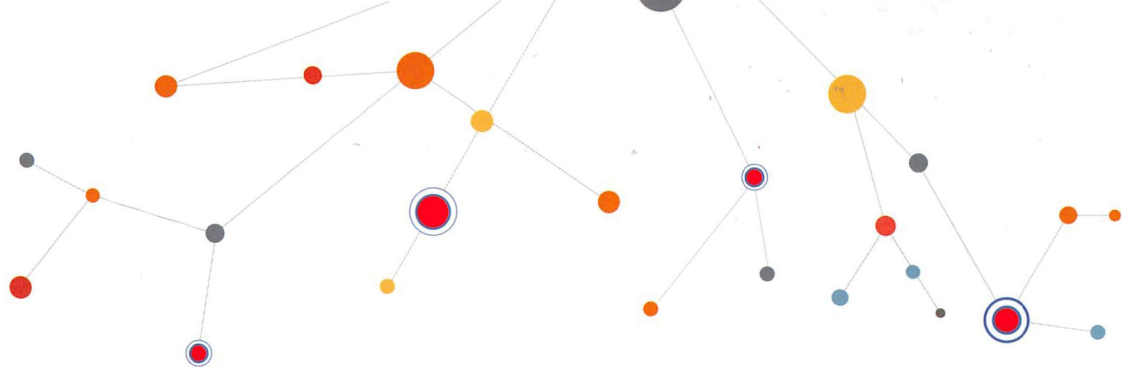
知名专家联袂推荐，实力专家联合撰写，全面性、透彻性毋庸置疑。深度讲解区块链核心技术、平台与应用开发，涵盖架构、共识、加密、P2P、比特币、以太坊、Hyperledger、EOS、潜力框架、问题与测评等。本书分为三篇，内容解读如下。

**基础篇（第1～6章）**，着重讲解区块链技术思想、通用架构和核心技术。该部分写作时注意通俗易懂且兼顾全局，是学习基石与蓝图，涵盖区块链思想与价值、通用架构模型、基础概念与核心技术（加密、共识、P2P网络等）。

**实战篇（第7～9章）**，讲解主流区块链开发平台（比特币、以太坊、Hyperledger Fabric）的核心机制、技术细节，并给出电子现金系统、智能合约开发、完整 Fabric 网络构建与应用三个案例。

**进阶篇（第10～12章）**，为进一步提升读者开发能力、眼界与研究方向，涵盖三个方面：

- ① 可能的发展方向，以及一些富有潜力、特色的区块链平台（EOS、Cardano、IOTA等）；
- ② 区块链开发需要考虑的各种问题，包括技术局限、各种安全问题与漏洞、应对措施；
- ③ 区块链测评，从6个层面和8大类质量指标来设计区块链项目测评点和测试用例。



邹均编写的本书，与其上一本著作《区块链技术指南》，以及市场上关于区块链的大量著作相比，以下几个方面值得注意：宏观视野、系统化、科学的描述方式、超越概念化、重视和引用英语文献对话。

——朱嘉明，中国数字资产研究院院长、著名经济学家

本书作者们本着求真务实的态度，力求给读者提供一本完整、翔实地反映当前区块链技术体系的书籍，是广大区块链爱好者和从业者可以参考学习的工具书，值得推荐。

——姚前，中国人民银行数字货币研究所所长

本书给读者一个全面系统地理解区块链的机会，既没有夸大，也没有贬低，观点比较客观中肯，值得推荐。

——斯雪明，复旦大学教授、中国计算机协会区块链专委会主任



投稿热线: (010) 88379604  
客服热线: (010) 88379426 88361066  
购书热线: (010) 68326294 88379649 68995259

华章网站: [www.hzbook.com](http://www.hzbook.com)  
网上购书: [www.china-pub.com](http://www.china-pub.com)  
数字阅读: [www.hzmedia.com.cn](http://www.hzmedia.com.cn)

上架指导: 计算机\区块链

ISBN 978-7-111-60614-7



9 787111 606147

定价: 99.00元